Virtualisation en contexte HPC

Antoine Capra

Thèse présentée
pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

École doctorale de Mathématiques et Informatique de Bordeaux
Spécialité : **Informatique**

Par **Antoine CAPRA**

Virtualisation en contexte HPC

Soutenue le 17 décembre 2015

Membres du jury :

M. J.-M. Pierson	Professeur, Université de Toulouse	Rapporteur
M. G. Thomas	Professeur, Telecom Sud-Paris	Rapporteur
M. C. Consel	Professeur, Université de Bordeaux	Président du jury
M. F. Diakhaté	Ingénieur-chercheur au CEA, Docteur	Encadrant CEA
M. R. Namyst	Professeur, Université de Bordeaux	Directeur de thèse
M. M. Pérache	Ingénieur-chercheur au CEA, HDR	Encadrant CEA
MME. P. Rossé-Laurent	Ingénieur-chercheur ATOS, Docteur	Examinateur

Résumé

Afin de répondre aux besoins croissants de la simulation numérique et de rester à la pointe de la technologie, les supercalculateurs doivent d'être constamment améliorés. Ces améliorations peuvent être d'ordre matériel ou logiciel. Cela force les applications à s'adapter à un nouvel environnement de programmation au fil de son développement. Il devient alors nécessaire de se poser la question de la pérennité des applications et de leur portabilité d'une machine à une autre. L'utilisation de machines virtuelles peut être une première réponse à ce besoin de pérennisation en stabilisant les environnements de programmation. Grâce à la virtualisation, une application peut être développée au sein d'un environnement figé, sans être directement impactée par l'environnement présent sur une machine physique. Pour autant, l'abstraction supplémentaire induite par les machines virtuelles entraine en pratique une perte de performance. Nous proposons dans cette thèse un ensemble d'outils et de techniques afin de permettre l'utilisation de machines virtuelles en contexte HPC. Tout d'abord nous montrons qu'il est possible d'optimiser le fonctionnement d'un hyperviseur afin de répondre le plus fidèlement aux contraintes du HPC que sont : le placement des fils d'exécution et la localité mémoire des données. Puis en s'appuyant sur ce résultat, nous avons proposé un service de partitionnement des ressources d'un nœud de calcul par le biais des machines virtuelles. Enfin, pour étendre nos travaux à une utilisation pour des applications MPI, nous avons étudié les solutions et performances réseau d'une machine virtuelle.

Mots clés : Calcul haute performance, Virtualisation, MPI, OpenMP

Cette thèse a été préparée au CEA, DAM, DIF, F-91297 Arpajon, France.

Abstract

To meet the growing needs of the digital simulation and remain at the forefront of technology, supercomputers must be constantly improved. These improvements can be hardware or software order. This forces the application to adapt to a new programming environment throughout its development. It then becomes necessary to raise the question of the sustainability of applications and portability from one machine to another. The use of virtual machines may be a first response to this need for sustaining stabilizing programming environments. With virtualization, applications can be developed in a fixed environment, without being directly impacted by the current environment on a physical machine. However, the additional abstraction induced by virtual machines in practice leads to a loss of performance. We propose in this thesis a set of tools and techniques to enable the use of virtual machines in HPC context. First we show that it is possible to optimize the operation of a hypervisor to respond accurately to the constraints of HPC that are : the placement of implementing son and memory data locality. Then, based on this, we have proposed a resource partitioning service from a compute node through virtual machines. Finally, to expand our work to use for MPI applications, we studied the network solutions and performance of a virtual machine.

Key words : High Performance Computing, Virtualisation, MPI, OpenMP

Remerciements

Je remercie tout d'abord mon directeur de thèse Raymond Namyst ainsi que mes encadrants CEA : François Diakhaté et Marc Pérache pour m'avoir guidé pendant ces trois années. Ce manuscrit n'aurait pas vu le jour sans votre confiance et vos conseils avisés. Je serai ravi de pouvoir de nouveau travailler avec vous.

J'adresse mes remerciements à mes rapporteurs Jean-Marc Pierson et Gaël Thomas pour la lecture de mon manuscrit et leurs judicieuses remarques afin de l'améliorer. Merci à Charles Consel et Pascale Rossé-Laurent d'avoir accepté de participer à mon jury en qualité de président de jury et examinateur.

Merci à «ma binôme» Emmanuelle , j'ai pris un grand plaisir à travailler avec toi que ce soit dans nos projets communs en master ou dans nos discussions pendant ces trois ans de thèse. Merci à Jérôme Clet-Ortega pour nos courses à pieds le week-end ainsi que nos petits délires musicaux que seuls quelques initiés peuvent comprendre m'ont permis de souffler et d'apprécier au mieux ces trois intenses années. Merci à Adrien pour m'avoir fait découvrir la Corse et Nancy, je tacherai à l'avenir d'être plus organisé ! Merci à Gautier pour ses histoires aussi improbables que drôles qui ont pimentées nos pauses cafés pendant ma dernière année.

Je remercie également le personnel CEA ainsi que les doctorants, CEA ou non, que j'ai pu croiser tout au long de ces années, vous avez su m'apporter un regard neuf sur le domaine de la simulation numérique et du calcul haute performance. Je ne me risquerai pas à lister vos noms par peur de froisser l'un d'entre vous par un oubli. Les souvenirs que nous partageons sont autant de bons moments dont je prends plaisir à me rappeler et à raconter. Je souhaite aux doctorants qui vont me succéder bon courage et plein de réussite pour les prochaines années.

Merci à Sameer Shende et Allen Malony ainsi que Jean-Baptiste Besnard qui m'ont accordées leur confiance en me permettant de continuer à évoluer dans la sphère du HPC. Merci à mon co-bureau Julien Adam pour ses encouragements, sa bonne humeur quotidienne et sa maitrise de Git !

Merci à ma famille qui a toujours su me soutenir dans les périodes de joie comme celles de déprime. Je remercie tout particulièrement mes parents pour toutes ces dernières années, vous avez su m'apporter ce dont j'avais besoin pour m'épanouir au quotidien.

Enfin, je remercie ma chère et tendre Émilie dont la présence, la patience et les sentiments font de moi chaque jour un homme heureux. Tu as su m'apporter tout le soutien nécessaire pour mener à besoin la rédaction de ce manuscrit et mes travaux de recherche.

Table des matières

Table des figures

Introduction

De tout temps, l'homme a cherché à comprendre le fonctionnement et les règles qui régissent le monde qui l'entoure. Pour faire progresser les connaissances scientifiques, il se base sur l'observation des phénomènes naturels et tente de les formaliser sous forme de modèles. Ces modèles sont en général des abstractions plus ou moins fines du phénomène réel que le chercheur souhaite étudier. Les abstractions sont ensuite traduites en système d'équations dont on confronte les solutions approchées ou exactes aux données expérimentales. Dans la pratique, les équations mathématiques d'un système ne peuvent être résolues de manière analytique et sont résolues de manière approchée à l'aide de méthodes itératives jusqu'à la convergence de la solution numérique. Même si le terme de simulation numérique n'est pas connu, les simulations numériques étaient réalisées bien avant les ordinateurs. Au XVIIIe siècle, les calculs de Laplace effectués à la main en s'aidant des tables numériques (logarithmes, racines...) pour résoudre son modèle planétaire simplifié peuvent être considérés comme une simulation numérique. L'ajout de nouveaux paramètres ou d'interactions entre les planètes au modèle planétaire de Laplace rend le calcul manuel des solutions irréalisables dans un temps raisonnable. Ce constat est commun à de nombreux modèles physiques dont les méthodes de résolution demandent un nombre important de calculs pour être résolues, ce processus est alors automatisé sous la forme d'un programme informatique puis exécuté sur un ordinateur. Plus le système est proche de la réalité, plus le calcul sera complexe. Plus il est complexe, plus le programme sera long à s'exécuter. C'est donc le besoin permanent de résultat toujours plus précis en un temps raisonnable qui explique l'avènement de la simulation numérique sur ordinateur parallèle à tel point que la simulation numérique est devenue aujourd'hui tributaire de l'outil informatique.

Pour répondre aux besoins de la simulation numérique, l'informatique a dû évoluer afin de proposer des ordinateurs de plus en plus puissants ainsi que des langages permettant d'écrire des modèles mathématiques sous forme de programme. Des premiers ordinateurs mainframes aux supercalculateurs modernes, l'informatique dédiée à la simulation numérique ou calcul haute performance a connu un grand nombre de révolutions. Les calculateurs sont dans un premier temps des ressources rares et centralisées réservées à un usage académique. L'apparition de l'ordinateur personnel dans les années 90 ainsi que l'amélioration des réseaux d'interconnexion va permettre l'avènement des supercalculateurs tels que nous les connaissons aujourd'hui : les grappes de calcul. Ces énormes machines composées de centaines de milliers de processeurs interconnectés sont aujourd'hui capables de réaliser des simulations rivalisant avec la qualité de certaines expériences pour des coûts dérisoires par rapport à celles-ci. De plus, le HPC permet de réaliser sans risque l'étude de phénomène pouvant être expérimenté de manière concrète (design d'une aile d'avion, crash test automobile...), mais aussi non observable comme la simulation de l'expansion de l'univers depuis le Big-Bang (Projet DEUS). Le HPC est donc aujourd'hui devenu une composante essentielle à la recherche académique et industrielle en permettant l'étude sans risque et peu coûteuse de phénomènes observables ou non.

Pour autant, la simulation numérique n'est plus le seul domaine ayant un fort besoin de puissance de calcul. En effet, l'essor des systèmes informatiques et la production de données numériques entrainent de facto le besoin d'infrastructure spécialisée. C'est à dire capable de stocker, accéder et mettre en relation d'importants volumes de données ou de statistiques. Ce nouveau champ de recherche appelé Big-Data présente des caractéristiques différentes des applications scientifiques classiques. Leur performance n'étant plus seulement liée aux débits d'opérations arithmétiques par seconde, mais aux débits d'accès aux données par unité de temps. C'est le cas par exemple de l'étude du génome humain dont la baisse du coût du séquençage de l'ADN a permis des avancées majeures en génomique et biochimie. Il devient alors possible de comparer l'ADN de deux populations, une saine et une malade afin d'isoler le gène responsable de la maladie. Ce traitement nécessite d'importants moyens de calcul et la manipulation d'énormes volumes de données. Les supercalculateurs doivent donc s'attaquer à de nouveaux défis afin de répondre aux besoins hétérogènes des applications : HPC et Big-Data.

L'augmentation de la puissance de calcul a entrainé l'élaboration de machines de plus en plus complexes de par l'organisation hiérarchique de leurs composants : processeurs, mémoires et infrastructures réseau. Cette complexité a une forte incidence sur les choix de programmation d'une application scientifique. En effet, il n'est pas possible d'exploiter finement une machine parallèle sans prendre en compte l'organisation générale de la machine. Pour cela, l'utilisateur se base sur des modèles de programmation ainsi qu'un ou plusieurs supports exécutifs parallèles. Le modèle de programmation lui fournit une syntaxe permettant d'exprimer le degré de parallélisme au sein de son application. Il est ensuite à la charge du support exécutif de répartir et d'ordonnancer le traitement de l'application en fonction des informations fournies par l'utilisateur. La performance d'une application parallèle est donc régie par le degré de précision de la syntaxe du modèle de programmation utilisé, mais aussi du fonctionnement du support exécutif. En pratique, il est très complexe, même à jeu d'entrée identique, de prédire le comportement d'une application parallèle qui par nature est localement voire globalement indéterministe. Il est donc nécessaire de fournir aux programmeurs tous les outils nécessaires à une exécution efficace de son application.

De plus, une application n'est pas simplement un code informatique indépendant. Elle a besoin pour fonctionner d'outils et de bibliothèques génériques qui lui permettent de ne pas être intégralement liée à la machine sur laquelle le code est développé. Cependant ces outils et bibliothèques qui forment un environnement de programmation participent à l'exécution du programme. Leurs modifications sont donc sont susceptibles de modifier le comportement de l'application selon leurs versions et implémentations. Le remplacement d'une machine entraine une mise à jour de cet environnement afin de l'adapter aux caractéristiques de celle-ci. Les machines ont aujourd'hui un cycle de vie inférieur à celui du développement d'une application et l'adaptation d'une application à un nouvel environnement de programmation est coûteuse en temps et en argent. Il devient alors nécessaire de se poser la question de la pérennité des applications et de leur portabilité d'une machine à une autre.

Les systèmes informatiques doivent donc aujourd'hui s'adapter à une demande variée aussi bien en terme de classes d'utilisateurs qu'en terme d'usages. Pour faire face à cette hétérogénéité, la virtualisation s'est imposée dans des domaines variés tels que la consolidation de serveurs ou l'informatique en nuage (cloud-computing). La virtualisation consiste à faire fonctionner un système d'exploitation comme un simple logiciel. Ce procédé permet à plusieurs systèmes d'exploitation d'utiliser une même machine physique par le biais d'un hyperviseur. Un hyperviseur peut s'exécuter directement sur

le matériel ou de manière logicielle au-dessus du système d'exploitation d'une machine physique. Cette caractéristique ajoute un niveau d'abstraction permettant de fournir des environnements variés sans modifier la configuration d'une machine, facilitant ainsi l'administration d'un cluster.

Cadre de thèse et contribution

L'utilisation de machines virtuelles peut être une première source de pérennisation d'un environnement de programmation. La virtualisation permet aux environnements de programmation d'être moins dépendants d'une architecture de machine particulière. Grâce à la virtualisation, ce n'est plus le code qui s'adapte à la machine, mais c'est la machine virtuelle qui s'adapte à la nouvelle architecture matérielle. L'application scientifique développée n'a plus à être modifiée à chaque changement d'environnement comme c'est le cas aujourd'hui. Pour autant, l'abstraction supplémentaire induite par les machines virtuelles entraine en pratique une perte de performance lors de l'exécution séquentielle ou parallèle d'un programme. De plus, les systèmes d'exploitation invités n'ont pas un accès direct aux périphériques physiques sous-jacents, notamment réseau, qui sont sous le contrôle du système d'exploitation hôte. Les communications réseau entre les machines virtuelles doivent être adaptées pour respecter les contraintes de performances inhérentes au HPC.

Les travaux présentés dans cette thèse se situent dans ce contexte. Nous proposons aux développeurs d'application un outil de déploiement d'environnements personnalisés gérés de façon dynamique. Pour cela, nous avons réalisé un lanceur de machines virtuelles capable de configurer des environnements virtuels personnalisés (noyau, bibliothèques...) en fonction des ressources allouées par le développeur. Le surcoût en temps d'exécution au sein de ces environnements virtuels optimisés d'application est de l'ordre de 2 à 3 % des performances d'un environnement standard. Ce surcoût est raisonnable et nous permet de construire un gestionnaire de ressources dynamiques pour applications OpenMP concurrentes s'exécutant dans des machines virtuelles. Notre gestionnaire de ressources se base sur une connaissance fine de la topologie matérielle et partitionne un nœud de calcul en préservant la localité mémoire. Enfin, nous montrons comment étendre l'usage des machines virtuelles au modèle de programmation MPI qui nécessite des communications entre machines virtuelles. L'objectif est de permettre un dimensionnement à la volée des ressources de chaque processus MPI et leur migration à l'aide de machines virtuelles. Ces caractéristiques deviennent essentielles à la gestion dynamique des ressources d'un supercalculateur afin de coller au plus proche des besoins de l'application.

Organisation du document

Le chapitre 1 est découpé en deux parties que sont : l'introduction au calcul haute performance et la virtualisation. Il présente dans un premier temps l'évolution des architectures matérielles des systèmes parallèles ainsi que la manière de les programmer. Puis dans un deuxième temps, il définit et décrit les techniques permettant l'exécution d'une machine virtuelle. Le chapitre 2 présente la démarche suivie pour optimiser les performances d'une application multithreadée exécutée au sein d'une machine virtuelle. Cette optimisation permet l'utilisation et la conception de services à base de machines virtuelles au sein de clusters HPC. Un premier service de gestion dynamique des ressources d'applications OpenMP concurrentes est présenté pour illustrer une utilisation possible des machines virtuelles. Le chapitre 3 décrit la mise en place d'une solution

réseau pour des machines virtuelles dédiées au calcul haute performance. Pour cela, il est présenté la mise en place d'un pilote au sein du support exécutif MPC. L'objectif est de permettre des communications efficaces entre deux processus MPI exécutés dans des machines virtuelles séparées. L'exécution d'instances MPI lancées de manière distribuée implique de les synchroniser lors de leur exécution au moyen d'un DNS. Ce mécanisme DNS permet un routage dynamique des messages entre machines virtuelles. Enfin, nous conclurons sur les travaux réalisés et discuterons sur les perspectives envisageables.

Chapitre 1

Introduction au calcul haute performance et à la virtualisation

"En informatique, la miniaturisation augmente la puissance de calcul. On peut être plus petit et plus intelligent. "

Bernard Werber, Les Micro-humains (2013)

1.1 Introduction au calcul haute performance

Dès les années 1940, l'électronique a fait des progrès suffisants pour permettre la conception des premiers calculateurs programmables. L'introduction du transistor en 1947 par Bell, l'invention de la microprogrammation par Maurice Wilkes en 1955 ou bien le premier microprocesseur par Intel en 1971 sont autant de révolutions qui ont permis l'avènement des supercalculateurs modernes tels que nous les connaissons aujourd'hui. Cette évolution des architectures est principalement motivée par les besoins croissants des applications scientifiques. Elle permet de réaliser des calculs de plus en plus complexes de manière plus précise ou plus rapide. La parallélisation figure parmi les techniques développées. Elle consiste à subdiviser un calcul en plusieurs opérations concurrents afin d'en accélérer le traitement. Cette technique a pris une place prépondérante dans le développement des machines de calculs actuelles, on parle alors d'architectures parallèles. L'exploitation fine et la conception de ces architectures sont communément appelées le calcul haute performance (HPC).

Dans ce chapitre, nous dressons un tour d'horizon du calcul haute performance qui constitue le cadre d'action de nos travaux. Nous détaillons l'évolution du calcul parallèle et plus spécifiquement celle des systèmes parallèles, des processeurs qui les composent ainsi que leur réseau d'interconnexion. Enfin, nous expliquons les paradigmes et concepts permettant l'utilisation et la programmation de ces systèmes parallèles.

1.1.1 Des mainframes aux supercalculateurs modernes

Les premiers systèmes parallèles ont d'abord été composés d'une seule unité centrale de traitement (CPU) comme c'est le cas pour le CDC6600[Tho80] livré pour la première fois en 1964. Le CDC6600 utilise, en plus de son CPU qui exécute le programme principal, dix processeurs périphériques permettant de réaliser les entrées/sorties des données. Il est capable de réaliser 3 Mflops à 40 MHz et possède une mémoire de 131000 mots de 60 bits, il n'utilise pas d'octet, mais 10 caractères de 6 bits par mot. Son processeur principal est superscalaire. Un ordinateur superscalaire est capable de réaliser certaines

instructions en concurrence, l'objectif étant d'exploiter le parallélisme entre instructions pour accélérer l'exécution d'un programme séquentiel ou parallèle de manière transparente. En 1976, le Cray-1[Rus78] offre des performances nettement améliorées avec 166 Mflops à 83 MHz et permet l'utilisation d'une mémoire de 8 Mo. Son architecture est de type vectoriel et lui permet de réaliser une opération sur un ensemble de nombres contigus (vecteurs) en mémoire, à l'inverse des opérations scalaires qui s'appliquent sur un seul nombre ou mot. En seulement huit ans, la fréquence a été multipliée par deux et le nombre d'instructions par seconde par un facteur de plus de cinquante.

Les architectures monoprocesseurs sont rapidement remplacées par des machines composées de plusieurs CPU identiques reliés à une mémoire partagée. On parle d'architecture *Symmetric MultiProcessor* (SMP). Le Cray-X/MP livré dès 1982 figure parmi les premières architectures SMP avec des configurations de deux ou quatre processeurs d'architecture équivalente à celle d'un Cray-1 pour un total de 800 Mflops. Si l'on compare la puissance du Cray-X/MP avec des processeurs d'aujourd'hui, il serait proche de la capacité de calcul d'un téléphone portable.

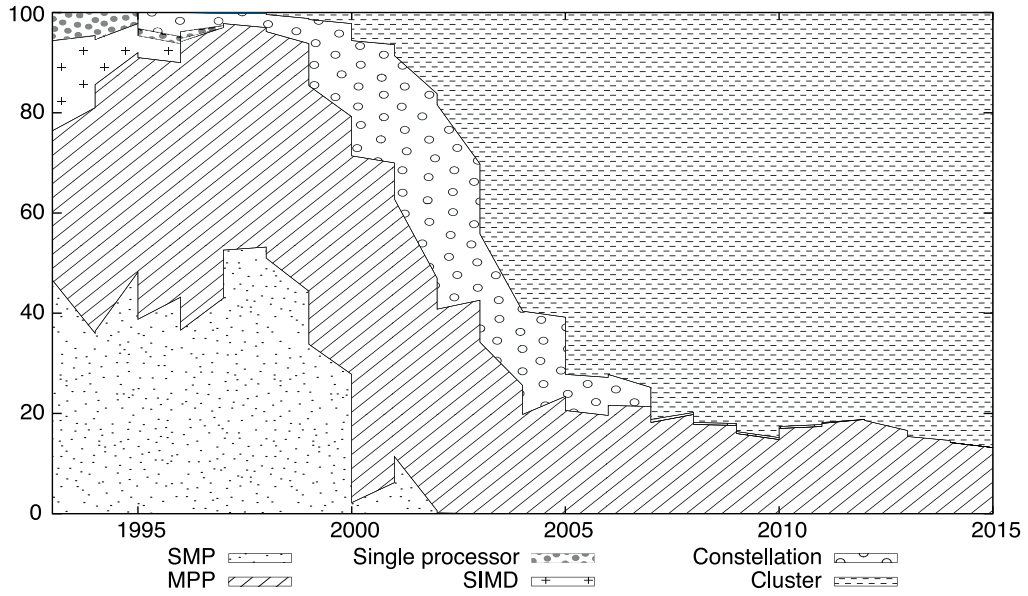


FIGURE 1.1 – Évolution des architectures (source Top500 - Juin 2015)

Les machines possédant un unique processeur multicœurs seront remplacées dès les années 2000 par des architectures plus complexes que sont les constellations, *Massively Parallel Processor* (MPP) et les grappes de calculs (cluster). Ces trois architectures se différencient par leur réseau d'interconnexion. La figure 1.1 illustre le pourcentage d'utilisation d'une architecture donnée au sein des machines de simulation en fonction des années. On peut observer que l'année 2000 marque la fin de la présence de machines monocœurs. On remarque aussi la montée en puissance et le succès des grappes de calcul dès les années 2002.

1.1.1.1 Le succès des grappes de calcul

En parallèle du développement des supercalculateurs dédiés au calcul scientifique, les ordinateurs à usage personnel ont peu à peu conquis les entreprises, les laboratoires et les foyers. Ces ordinateurs ont une capacité de calcul faible, mais ont l'avantage d'être peu onéreux. Ces machines peuvent être reliées par un réseau d'interconnexion, il est alors possible de cumuler la puissance de calcul de l'ensemble des machines d'un même

réseau. Ce système permet aussi d'exploiter des machines peu utilisées afin d'augmenter l'efficacité d'utilisation d'un parc de machines individuelles. C'est le cas du projet CONDOR[LML87] réalisé par l'université américaine du Wisconsin. L'université est dotée d'un parc informatique regroupant une centaine de machines dont la part d'utilisation des machines est de l'ordre de 30%. Ce constat montre un gâchis de ressources et le projet CONDOR vise à permettre l'exploitation des stations inutilisées. Ces réseaux de machines sont toutefois limités par la disponibilité des machines et la capacité du réseau d'interconnexion. Leurs capacités de calcul sont donc très loin des capacités d'un supercalculateur. Pour autant, les clusters représentent une alternative peu onéreuse pour les centres ne disposant pas de moyens financiers suffisants.

Les années 90 marquent l'essor de nouvelles solutions d'interconnexion plus efficaces et plus fiables : le débit est ainsi passé de quelques Mbit/s à plusieurs dizaines de Gbit/s et la latence a atteint la microseconde. Afin de maximiser les performances obtenues par ces technologies, les réseaux de stations sont regroupés physiquement ce qui a pour conséquence de minimiser les délais de transmission. Les stations sont alors uniquement dédiées aux calculs scientifiques aussi bien en terme de disponibilité qu'en terme de matériel. Ces nouveaux systèmes dédiés au calcul forment une nouvelle classe de système : les grappes de calcul.

1.1.2 Évolution de la puissance de calcul des supercalculateurs

L'évolution rapide que nous avons pu observer dans le bref historique s'explique en grande partie par les besoins toujours plus croissants de la simulation numérique. Réservée dans un premier temps à usage spécifique comme le secteur académique, la simulation numérique s'est aujourd'hui imposée comme un outil de conception indispensable pour l'industrie de tout horizon. De l'industrie automobile à l'industrie pétrolière en passant par les cosmétiques, la capacité de simulation est devenue un enjeu important. Pour comparer les machines et technologies, un classement mondial des machines est réalisé sur la base d'une application dénommée Linpack (HPL)[Don88].

1.1.2.1 Linpack (HPL)

Introduit par Jack Dongarra, le Linpack est un test de performance qui consiste en une résolution d'un système d'équations linéaires dense. Ce test de performance ou benchmark se doit d'être uniforme, il est donc implémenté avec la même méthode : LU avec pivot partiel de Gauss. Il possède un nombre d'opérations connu : $2/3n^3 + O(n^2)$ opérations flottantes à double précision. On peut donc en déduire la performance réelle d'une machine en nombre opérations flottantes par seconde (FLOPS) appelée Rmax. On utilise généralement deux autres métriques que sont : la performance maximale théorique (Rpeak) et l'efficacité (Rmax / Rpeak). Pour autant, le Linpack est utilisé dans le classement des machines les plus performantes depuis 1979.

1.1.2.2 Top500 & Green500

Le Top500 est une classification par ordre décroissant des 500 supercalculateurs les plus puissants au monde en terme d'opérations flottantes réalisées par seconde (Flops). Ce classement basé sur le Rmax d'un Linpack utilisant des nombres de 64 bits existe depuis 1993 et est mis à jour tous les six mois. La consommation électrique des supercalculateurs n'ayant pas cessé d'augmenter depuis ces 25 dernières années, l'efficacité énergétique et le développement durable sont devenus des enjeux majeurs. Pour promouvoir « l'informatique verte », le Green500[cFC07] classe les supercalculateurs du Top500[TOP] en mesurant leur flops/watt.

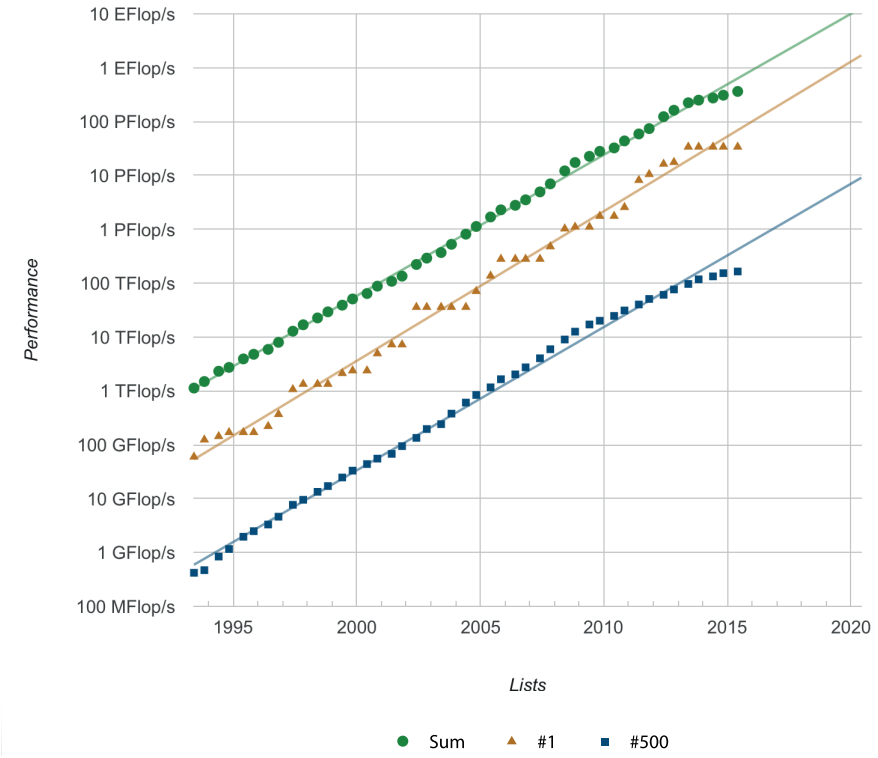


FIGURE 1.2 – Évolution des performances (source Top500 - Juin 2015)

> *"[...] we have reached a point where designing a system for good HPL performance can actually lead to design choices that are wrong for the real application mix, or add unnecessary components or complexity to the system."*
>
> JACK DONGARRA & MICHAEL A. HEROUX, 2013

Le Linpack est une application très régulière qui ne reflète pas en pratique le comportement standard d'une application numérique. De plus, le Linpack ne manipule que des nombres à virgule et ne donne pas d'information sur des codes purement réalisés sur des entiers. Le Top500 étant un indice de référence sur la puissance d'une machine, les constructeurs ont tout intérêt à construire des machines capables d'obtenir des résultats importants au test du Linpack. Pour Jack Dongarra, cet état de fait conduit à des choix architecturaux décorrélés des besoins des applications réelles. Le Top500 ne peut donc pas être utilisé comme seule référence d'un classement HPC.

1.1.2.3 Graph500 et HPCG

Pour offrir un spectre plus large des capacités d'une machine, d'autres benchmarks ont été introduits, parmi ces benchmarks on trouve Graph500 et HPCG.

Graph500 Graph500[US12], introduit en 2010 par Richard Murphy, est un benchmark dont l'objectif est de montrer les performances d'application de fouille et d'analyse d'importants ensembles de données. Afin de fouiller efficacement ces gigantesques silos de données, il faut qu'un supercalculateur possède des caractéristiques assez différentes de ceux qui exécutent des applications sur des nombres à virgule. Les graphes sont au cœur de tous les algorithmes de parcours de données ce qui explique ce choix dans la

Classe	Niveau	Échelle	Degré d'un sommet	Taille des données (To)
Toy	10	26	16	0,0172
Mini	11	29	16	0,1374
Small	12	32	16	1,1
Médium	13	36	16	17,6
Large	14	39	16	140,7
Huge	15	42	16	1125,9

FIGURE 1.3 – Classe de problème Graph500

spécification du benchmark Graph500. On parle ici de données structurées dont le volume varie entre une dizaine de gigaoctets et quelques Petaoctets. Comme le montre la figure 1.3, les classes de problèmes gardent des propriétés identiques concernant le degré de chaque sommet, mais varient en terme de volume de données à analyser. Graph500 reproduit le besoin de différents secteurs comme la cybersécurité : les grandes entreprises peuvent générer 15 milliards de logs par jour et l'analyse de ces logs nécessite un parcours complet des données générées ou encore la gestion des dossiers médicaux d'un hôpital. Seules trois machines du Graph500 sont aujourd'hui capables de réaliser l'exécution de problème de type large.

HPCG HPCG[HPC13] est une version de l'algorithme de gradient conjugué préconditionné permettant de reproduire de manière plus fidèle que le linpack les schémas d'exécutions, les accès mémoires et les communications d'une application de simulation numérique. Le gradient conjugué est une méthode exacte qui converge en n itérations et permettant de résoudre des systèmes d'équations linéaires dont la matrice est symétrique définie positive. Le gradient conjugué est généralement utilisé comme une méthode itérative pour calculer un minimum ou maximum d'une fonction que l'on souhaite voir converger rapidement. Pour accélérer la vitesse de convergence, on peut préconditionner le système en multipliant le système initial par une matrice afin d'obtenir un nouveau système. La méthode du gradient conjugué appliquée à ce nouveau système permet une convergence plus rapide que le système initial.

Machine	Places			Pflop/s		
	Top500	Graph500	HPCG	Top500	HPCG	Ratio
Tianhe-2	1	N/A	1	33,9	0,58	1,7
Mira	3	3	4	8,6	0.17	1,9
K computer	4	1	2	10,5	0,46	4,4
Curie	44	N/A	17	1,36	0,05	3,7

FIGURE 1.4 – Performances et benchmarks de références (Juin 2015)

Les benchmarks Graph500 et HPCG ne bénéficiant pas de la même visibilité, il n'est pas possible de comparer directement les résultats des trois benchmarks. Toutefois, on

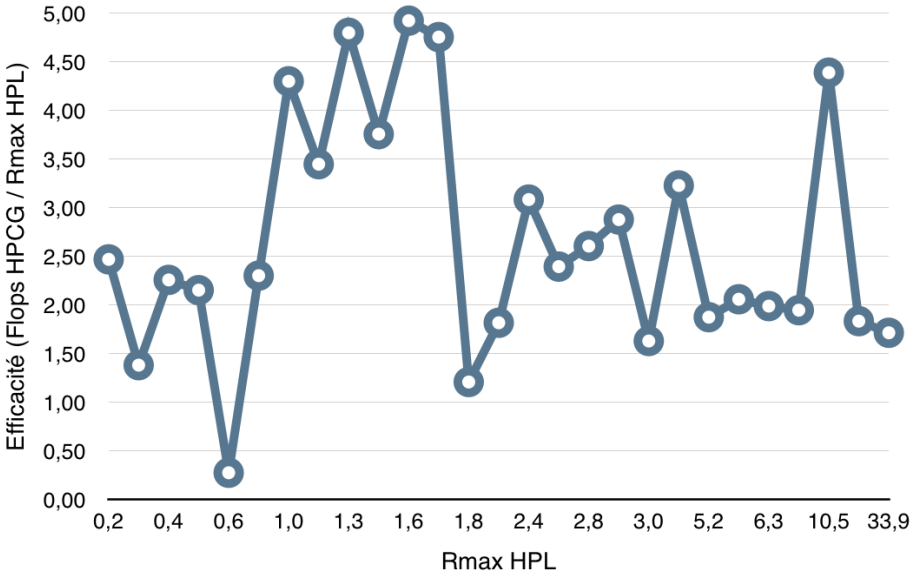


FIGURE 1.5 – Pourcentage de la crête exploitée en fonction de Rmax (HPL)

peut réaliser une analyse relative des statistiques de chacun de ces classements. Pour cela, on a recensé les performances de quatre machines du Top500. Il est à noter que Graph500 ne fournit pas de statistique en fréquence d'instructions par seconde, mais en fréquence de sommets parcourus par seconde. On observe sur la figure 1.4 que les classements peuvent modifier l'ordre du palmarès : le K computer, moins bon au Top500 que Mira, est meilleur dans les autres référentiels. De plus, la différence en nombre d'instructions par seconde pour ces deux mêmes machines est de l'ordre de 22 % en faveur de la machine Mira pour le Linpack, mais de l'ordre de 300 % au profit du K computer pour le test HPCG. De la même manière, on peut calculer le pourcentage de la crête calculé avec le Linpack lors de l'exécution du benchmark HPCG (cf. la colonne ratio de la figure 1.4). Nous avons tracé sur la figure 1.5, la proportion de la crête Rmax lors de l'exécution de benchmark HPCG. La crête ou capacité maximum d'instruction par seconde est utilisée à environ 3 % lors de l'exécution du code HPCG.

1.1.2.4 France, Europe et HPC

L'Allemagne, le Royaume-Uni et la France occupent respectivement les places quatre, cinq et six et disposent de 24,5% de la puissance de calcul mondial selon le dernier classement du Top500 paru en juin 2015. Les profondes disparités observées en fonction du référentiel choisi montrent que l'on ne peut pas résumer le HPC à une machine. La puissance de calcul d'un pays, véritable enjeu économique et industriel, ne peut donc pas être résumée au Top500. Elle se découpe en plusieurs domaines que sont la conception de machines puissantes, le développement d'applications parallèles efficaces ainsi que l'accès à des ressources de calcul. Dans ce domaine, la France, et plus généralement l'Europe, a su se doter de structures afin de mettre à disposition des ressources de calcul pour les chercheurs académiques et les industriels. C'est le cas du projet PRACE qui permet le partage de supercalculateurs de capacités et d'architectures différentes, la France y contribue notamment en proposant l'accès au supercalculateur Curie. Curie est installé au très grand centre de calcul (TGCC) de Bruyère-le-Châtel et peut réaliser jusqu'à 2 Pflops. Selon le site du grand équipement national pour le calcul intensif (GENCI), la France est très bien positionnée dans l'utilisation du projet PRACE. Elle est le premier pays en nombre de projets scientifiques retenus, mais également le premier pays en nombre d'industriels (grand compte et PME) qui utilisent depuis mi-2012 les

ressources PRACE.

Machine	**Pays**	**Marque**	**CPU**	**Cœurs**	**Top500**	**Rmax**
JUQUEEN	Allemagne	IBM	Power	458.752	9	5,0
SuperMUC	Allemagne	IBM	Xeon	147.456	20	2,9
HORNET	Allemagne	Cray	Xeon	94.608	23	2,7
FERMI	Italie	IBM	Power	163.840	32	1,79
CURIE	France	Bull	Xeon	77.184	44	1,36
MareNostrum	Espagne	IBM	Xeon	48.896	77	0,92

FIGURE 1.6 – Supercalculateur Petaflopique PRACE (Juin-2015)

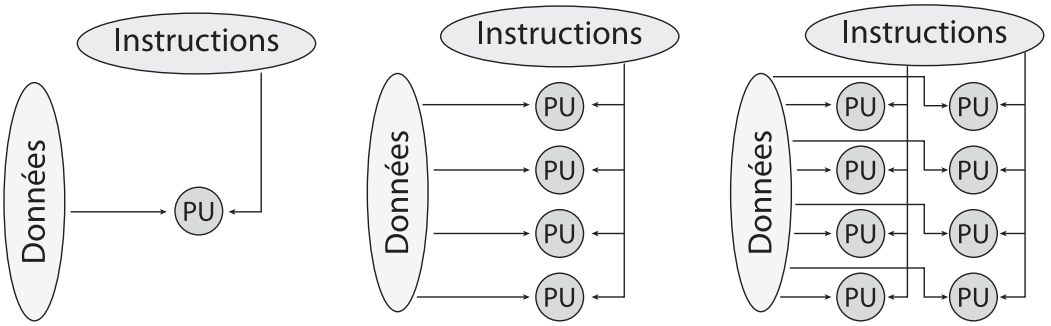
1.1.3 Architecture des ordinateurs

Nous avons vu qu'une grappe de calcul moderne est composée d'ilots, de nœuds et d'un réseau d'interconnexion. Nous allons maintenant décrire plus finement les différents éléments composant un nœud de calcul.

1.1.3.1 Taxonomie de Flynn

La taxonomie de Flynn[Fly11] est une classification des architectures d'ordinateurs proposée par Michael Flynn en 1966. Cette classification définit quatre types d'organisation du flux de données et du flux d'instructions. Les catégories se basent sur la nature unique ou multiple pour chacun des flux dont seuls trois ont une existence concrète.

— SISD (instruction simple, donnée unique) correspond à une architecture séquentielle sans parallélisme aussi bien en terme de mémoire que d'instruction.
— SIMD (instruction simple, données multiples) correspond à une architecture vectorielle c'est-à-dire possédant un parallélisme de mémoire.
— MIMD (instructions multiples, données multiples) correspond à une architecture multiprocesseur à mémoire distincte.



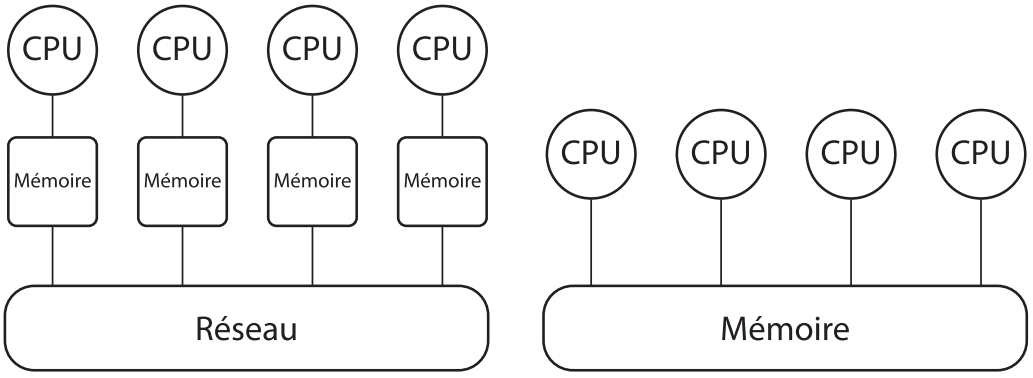
(a) SISD (b) SIMD (c) MIMD

FIGURE 1.7 – Taxonomie de Flynn

1.1.3.2 Mémoire distribuée et mémoire partagée

Dans le cas d'une mémoire partagée, tous les processeurs ont accès à la même mémoire. Les synchronisations entre les processeurs peuvent donc être réalisées de manière

implicite à l'aide de cette mémoire commune. La mise à jour d'une zone mémoire est vue de manière transparente pour les autres processeurs. À l'inverse dans le cas d'une architecture distribuée, les processeurs n'ont pas tous accès à la même mémoire, il est donc nécessaire d'utiliser des mécanismes de synchronisation et de communication.



(a) Mémoire distribué (b) Mémoire partagée

FIGURE 1.8 – Catégorie de mémoire

Mémoire physique distribuée La figure 1.8a illustre le cas d'une mémoire distribuée, chaque CPU a un accès à sa propre mémoire et ne peut directement accéder à la mémoire d'un autre CPU. Les synchronisations entre CPU, qu'elles soient en temps ou en donnée, sont réalisées de manière explicite. On utilise généralement dans ce cas un protocole d'échange de messages. Le principal avantage de cette architecture est l'absence de concurrence lors d'un accès à la mémoire, mais cela nécessite un échange de message au moyen d'un lien de communication réseau. L'accès à une mémoire distante entraine donc une latence et un débit plus important que des accès à la mémoire locale du CPU.

Mémoire physique partagée À l'inverse, on peut construire des processeurs exposant une mémoire commune pour l'ensemble des CPU. Les données ne sont alors plus répliquées, mais mutualisées, les synchronisations sont alors implicites. En théorie, la modification d'une donnée est visible de manière transparente pour les autres CPU sans synchronisation. Dans la pratique, la modification et la lecture d'une donnée ne sont pas atomiques. C'est-à-dire que la modification d'une valeur n'est pas réalisée en une seule instruction cela entraine une latence entre l'accès à la valeur courante d'une donnée et sa mise à jour par un CPU. Ce comportement peut entrainer des comportements indéterministes selon l'ordre dans lequel des CPU exécutent leurs instructions. En ce qui concerne les performances, la mémoire partagée permet d'obtenir une bande passante et des débits optimaux entre les CPU. L'augmentation du nombre de CPU ayant accès à une même mémoire peut en pratique entrainer un trafic important et dégrader la bande passante du lien vers la mémoire. La carte mère, support de liaison (ou matrice) entre la mémoire et un processeur étant un support en 2D, il devient impossible de garantir un accès uniforme à la mémoire quand le nombre de CPU augmente. Certains CPU étant de facto localisés plus proches d'une mémoire que les autres, on parle d'accès UMA lorsque les CPU ont un accès uniforme à la mémoire et d'accès NUMA dans le cas non uniforme. Le système NUMA est conçu pour pallier les limites de l'architecture SMP dans laquelle tout l'espace mémoire est accessible par un unique bus. Les accès concurrents sont donc problématiques et viennent lourdement impacter la latence et la bande passante de la RAM. Une architecture plus adaptée devient donc nécessaire pour les systèmes ayant de nombreux processeurs. NUMA représente une position médiane entre

le SMP et le clustering (diverses machines). Les mémoires NUMA accroissent le besoin de localité des données lors de l'exécution d'un programme, l'accès à une donnée étant plus couteux selon la localité de la donnée : sur la mémoire la plus proche du CPU ou dans la mémoire la plus éloignée. Dans la suite du manuscrit, nous considérerons exclusivement les architectures de type NUMA, car elles correspondent au matériel à notre disposition. En outre, le problème des architectures UMA peut être inclus dans celui des problèmes liés à une architecture NUMA, les contraintes inhérentes aux architectures NUMA étant plus forte que pour les architectures UMA.

Mémoire virtuellement distribuée Comme nous l'avons décrit précédemment, la mémoire peut être distribuée physiquement, elle peut être aussi distribuée virtuellement. C'est-à-dire qu'elle est physiquement partagée, mais les CPU exécutent des acteurs parallèles concurrents qui ne partagent pas implicitement leur mémoire. C'est le cas lorsque l'on utilise une programmation à base de processus, chaque processus ne pouvant pas directement accéder à la mémoire d'un autre processus. Les processus peuvent exécuter un ou plusieurs fils d'exécution ou threads, ces acteurs parallèles concurrents partagent l'accès à la mémoire virtuelle d'un processus.

Mémoire virtuellement partagée Une mémoire partagée peut être émulée afin de faire croire à des processus qu'ils partagent un même espace d'adressage global, c'est le cas dans le modèle de programmation parallèle PGAS. On suppose que l'espace d'adressage de la mémoire globale est partitionné logiquement entre les processus, une partie de cette mémoire étant locale à un processus ou thread. Les PGAS sont à la base des modèles suivants : Unified Parallel C [UPC05], Coarray Fortran [NR98], Fortress [ACH+07], et SHMEM [CCP+10].

1.1.3.3 Architecture d'un nœud de calcul

Nous présentons ici des exemples de nœuds de calcul modernes présents sur la machine Curie du CCRT. Un nœud de calcul est composé d'une mémoire NUMA et possède quatre sockets de processeurs de huit cœurs comme le montre la figure 1.9. Les processeurs partagent quatre bancs mémoire NUMA, chacun étant local à chaque socket. Les accès réalisés par un CPU d'un socket vers la mémoire d'un autre socket sont plus lents que l'accès à la mémoire de son socket. Ces nœuds étaient ceux disponibles avant leur transformation en nœuds BCS (Bull Coherent Switch) de 128 cœurs.

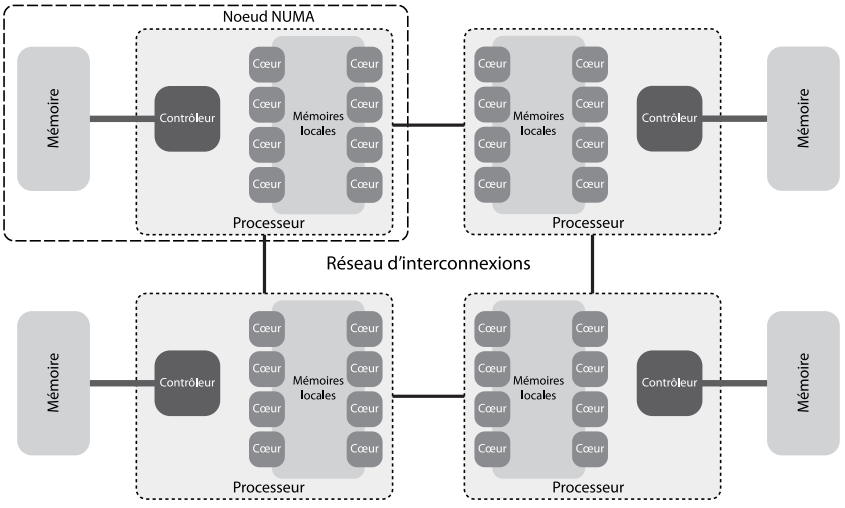


FIGURE 1.9 – Nœuds larges 32 cœurs

L'objectif premier des nœuds BCS est de répondre aux besoins importants en mémoire de codes de simulation particuliers comme le séquençage ADN. Les nœuds BCS sont constitués d'un niveau de plus que les nœuds larges précédents, plusieurs cartes mères des précédents nœuds larges étant connectés quatre par quatre. La sensibilité à la localité d'un nœud BCS est plus forte que sur les nœuds standards. La figure 1.10 montre la répartition des composants d'un nœud BCS à 128 cœurs.

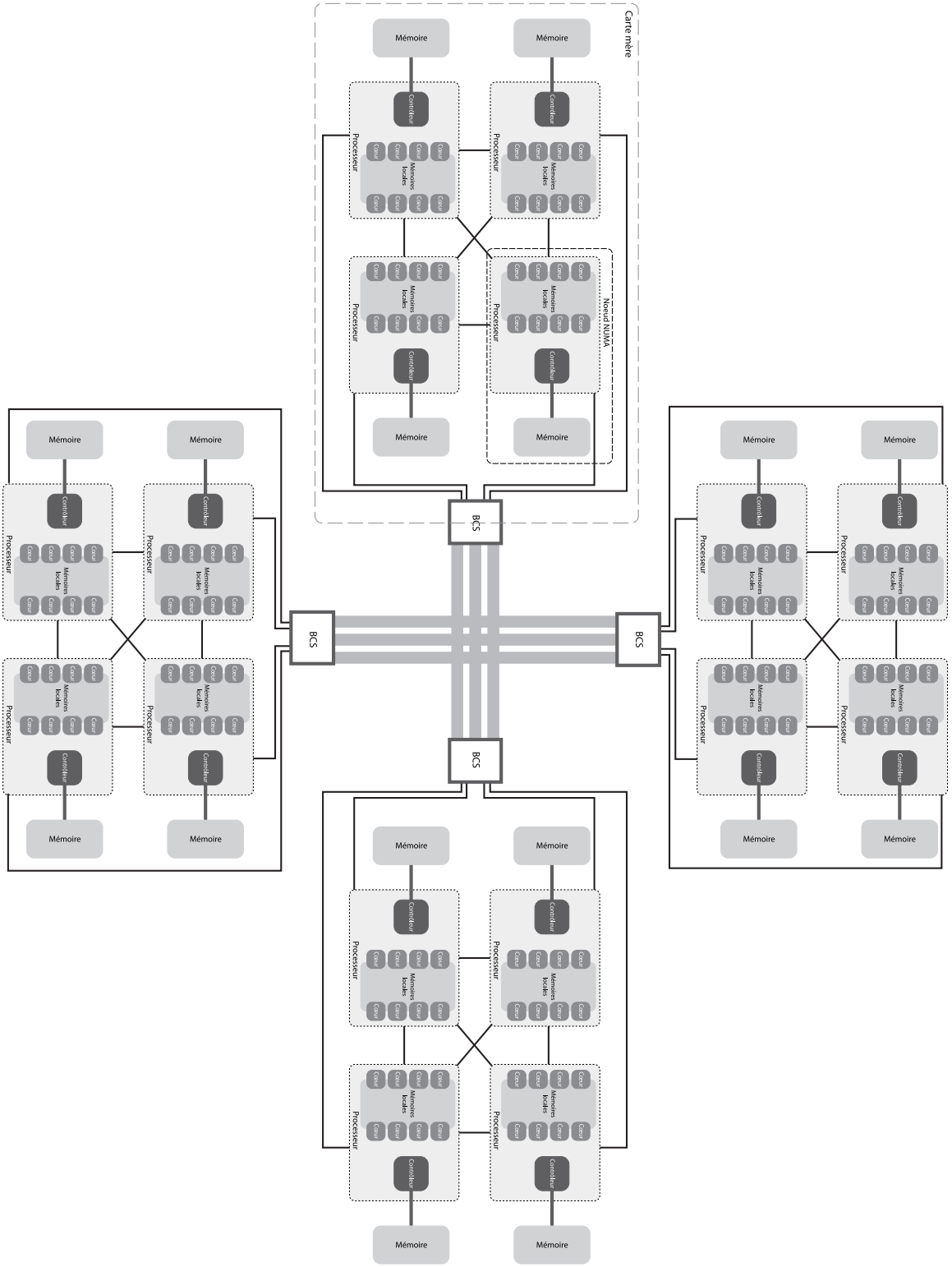


FIGURE 1.10 – Nœud BCS du supercalculateur Curie

Nous avons décrit les différents types d'exécutions et de hiérarchies mémoire d'un nœud de calcul, nous allons maintenant décrire comment programmer ces architectures hiérarchiques complexes.

1.2 Modèle de programmation

Nous présentons dans cette section les principaux modèles de programmation considérés comme standards par la communauté scientifique. Le parallélisme peut s'exprimer de plusieurs manières, on considère en général deux paradigmes : le parallélisme de tâches et le parallélisme de données. Chacun de ces paradigmes ayant ses propres supports exécutifs, avantages et limitations. Dans le cas d'une mémoire partagée, ces paradigmes utilisent le modèle *Fork/Join*.

1.2.1 Modèle *Fork/Join*

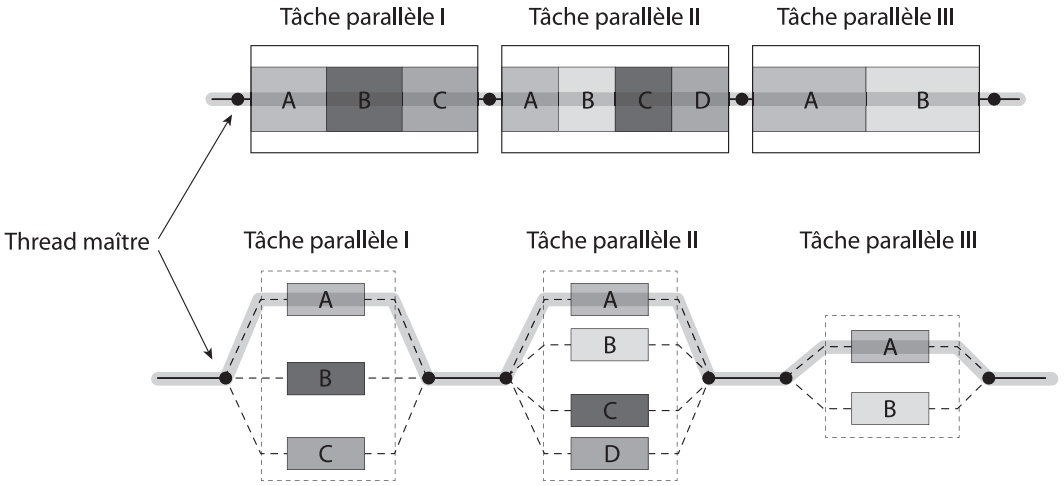


FIGURE 1.11 – Modèle *Fork/Join*

Le modèle de *Fork/Join* illustré dans la figure 1.11 est une manière de mettre en place une exécution parallèle d'un programme. L'exécution séquentielle du programme est illustrée en haut de l'image. Comme on peut le voir, le programme alterne des zones de branchement (*Fork*) en entrée de région dont les blocs sont exécutables en parallèle et *Join* quand l'exécution est séquentielle. Le modèle *Fork/Join* est un modèle d'exécution parallèle formulé dès les années 1963. Dans la pratique, le modèle *Fork/Join* est implémenté avec des processus légers (ou thread) en espace utilisateur contrôlé par un ordonnanceur capable de répartir les sections parallèles spécifiées par le programmeur.

Directives de parallélisation On peut utiliser des directives, c'est à dire des instructions à destination du compilateur, dans un code séquentiel. Ces directives indiquent comment peuvent être distribuées les données et les instructions sur les processeurs. Si le compilateur peut prendre en charge les directives qui lui sont transmises alors il se charge de la distribution des données, du calcul et des communications sinon les directives sont simplement ignorées. Cette approche nécessite l'utilisation d'une mémoire partagée entre les différents processus ou threads participant au calcul. On peut citer la bibliothèque HPF pour le langage Fortran, OpenMP[Ope08] pour les codes C, C++, Fortran ou OpenACC (Open Accelerators).

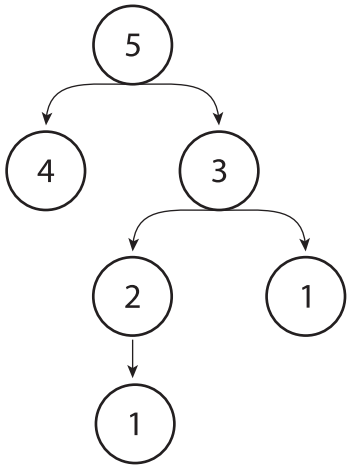
1.2.2 Parallélisme de tâches

Dans ce parallélisme, une tâche est une portion de code pouvant être divisée en sous-tâches indépendantes qui sont exécutées sur des unités de calcul distinctes. La réalisation de la tâche principale nécessite généralement des points de synchronisation ou d'échanges de données entre les sous-tâches. Ces communications limiteront généralement le gain de performance atteignable. La programmation par tâches est particulièrement adaptée pour paralléliser des problèmes irréguliers comme les boucles non bornées, les algorithmes récursifs et les schémas producteurs/consommateurs. Parmi les algorithmes qui se prêtent bien à l'utilisation du parallélisme de tâches, on peut citer l'algorithme de Fibonacci. La valeur au rang n de la suite de Fibonacci se calcule à l'aide des deux précédents rangs de ce la manière suivante :

$$f(0) = 0, f(1) = 1, f(n+2) = f(n+1) + f(n)$$

Suite de Fibonacci en OpenMP

```
1  int fibonacci(int n)
2  {
3     int fnm1, fnm1;
4     if ( n == 0 || n == 1 )
5          return(n);
6
7  #pragma omp task shared(fnm1)
8     fnm1 = comp_fib_numbers(n-1);
9  #pragma omp task shared(fnm2)
10    fnm2 = comp_fib_numbers(n-2);
11 #pragma omp taskwait
12
13    return fnm1 + fnm2;
```



(a) Calcul de N=5

FIGURE 1.12 – Suite de Fibonacci

Il existe de nombreuses implémentations du parallélisme de tâches parmi celles-ci, on trouve les bibliothèques logicielles OpenMP, Threading Building Blocks ou encore Cilk.

1.2.2.1 OpenMP

OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. OpenMP est basé sur l'utilisation de directives d'une bibliothèque logicielle. Les directives sont des instructions traitées par le préprocesseur à destination du compilateur. Ces directives ne sont appliquées que si le compilateur les supporte.

Parallélisation d'une boucle avec OpenMP

```
1 #pragma omp parallel
2 {
3     #pragma omp for
4     for(i=0; i<SIZE; i++)
5         A[i] = B[i] + C[i]
6 }
```

Dans une application OpenMP, le nombre de threads peut être proposé par l'utilisateur, mais ce choix reste à la liberté du support d'exécution. L'utilisateur utilise de manière transparente des threads en ajoutant à son code des régions parallèles, dont l'exécution peut être réalisée en concurrence entre des threads. Il peut ainsi distribuer les itérations d'une boucle *for* ou exécuter des sections de codes en parallèle. Afin de reproduire les mécanismes de sémaphore ou de verrou, l'API OpenMP permet de déclarer des régions ou des opérations critiques.

Enfin, OpenMP supporte l'utilisation d'un parallélisme imbriqué. Le parallélisme imbriqué est utile dans le cas d'utilisation de bibliothèques scientifiques qui n'ont a priori pas de moyen de savoir dans quel contexte sont réalisés les appels à leurs fonctions. Mais aussi dans la génération de tâches pouvant elles-mêmes être découpées en plus petites tâches.

1.2.2.2 Threading Building Blocks (TBB)

Threading Building Blocks (TBB) [Rei07] est une bibliothèque permettant une programmation parallèle évolutive pour des codes C++. TBB est implémenté dans les compilateurs Windows, gcc et icc, il n'est donc pas lié à un environnement logiciel particulier. Il supporte l'expression du parallélisme imbriquée, c'est-à-dire de tâches créant à nouveau des tâches comme c'est le cas dans la suite de Fibonacci. L'utilisateur spécifie par le biais de l'API des tâches spécifiques et laisse au support exécutif le soin de réaliser efficacement l'exécution de ces tâches. L'utilisateur a une vision haut niveau des tâches et évite une implémentation complexe à l'aide de threads.

1.2.2.3 Cilk

Suite de Fibonacci en cilk

```
1 cilk int fib(int n)
2 {
3     if (n < 2)
4         return n;
5     int x = cilk_spawn fib(n-1);
6     int y = cilk_spawn fib(n-2);
7     cilk_sync;
8     return x + y;
9 }
```

Cilk [ACH+07] est une extension des langages C et C++ permettant de rapidement tirer parti des architectures parallèles. L'extension se base sur deux mots clés que sont :

— `spawn` qui permet de créer une nouvelle tâche à exécuter ;
— `sync` qui permet d'attendre le traitement des tâches lancées précédemment.

1.2.3 Parallélisme de données

Principe Le parallélisme de données ou décomposition de domaine consiste à décomposer les données en un ensemble de parties de tailles homogènes ou non qui seront affectées aux différents processus. Le but étant d'utiliser des processeurs multicœurs, les processus sont associés à des processeurs différents et sur des nœuds qui peuvent être différents également. La décomposition de domaine suit le modèle SPMD, chaque processus exécute le même programme, mais sur des données différentes. La figure 1.13 illustre le cas d'un domaine décomposé en 4 sous-domaines. Les parties grisées sont des

parties répliquées du domaine, elles permettent de connaître les données à la frontière des sous-domaines voisins. Le parallélisme de tâches se prête bien aux algorithmes de différences finies ou d'éléments finis permettant de résoudre des équations aux dérivées partielles. Chaque processus calcule indépendamment sur une partie du domaine global et communique à chaque itération les données sur la frontière. On trouve principalement deux approches pour réaliser du parallélisme de données selon que les communications se font de manière implicite ou explicite. L'exemple le plus classique est celui du calcul de la somme de deux vecteurs par des cœurs différents. La décomposition de domaine est de facto utilisée dans les modèles à mémoire partagée comme c'est le cas pour MPI[For94].

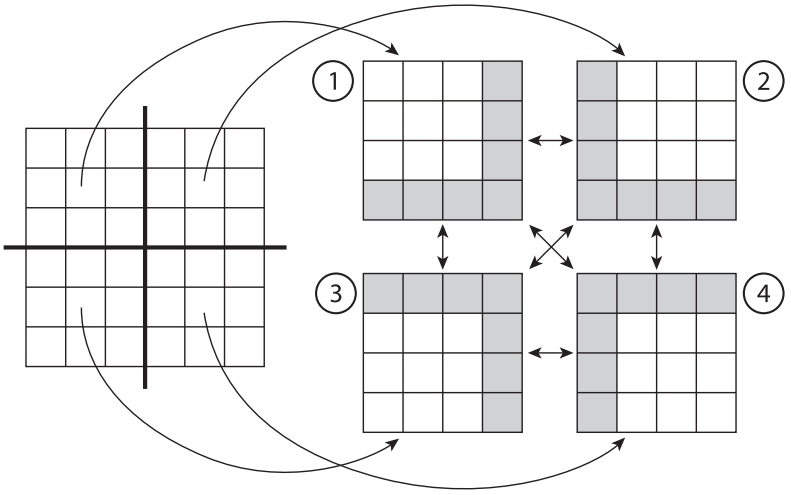


FIGURE 1.13 – Décomposition d'un domaine pour 4 processus

Somme de vecteurs entre processus MPI

```
1 MPI_Allreduce(B, A, SIZE, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
```

MPI : Message Passing Interface MPI est une norme introduite dans les années 1990 définissant une bibliothèque de fonctions permettant d'exploiter une mémoire distribuée par passage de message. MPI est devenue de facto un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. MPI possède l'avantage d'être grandement portable et permet d'obtenir de bonnes performances sur tout type d'architecture. MPI est basé sur trois concepts principaux : les communicateurs, les communications point-à-point et communications collectives. Un communicateur caractérise un ensemble de processus pouvant communiquer ensemble. Les communications point-à-point permettent à deux processus d'échanger des informations tandis que les communications collectives permettent l'exécution de fonctions pour tous les processus d'un même communicateur. Dans notre exemple, la fonction All_reduce permet à tous les processus d'obtenir le résultat de la somme des tableaux A locaux à chaque processus dans le tableau B.

Limitations Pour obtenir des performances intéressantes avec ce type d'algorithme, il faut que la taille de la partie locale au processus soit grande devant la taille de la frontière afin de limiter le nombre de communications nécessaires entre les différents processus. De plus, les temps de traitement du domaine local de chacun des processus doivent être suffisamment homogènes. Un processus n'ayant pas terminé le calcul de son domaine ne pourra pas communiquer les valeurs de sa frontière et sera attendu par tous les processus ayant besoin de ce résultat pour continuer. Par récursivité, on obtient que

les performances globales obtenues en utilisant un modèle en décomposition de domaine sont toujours limitées par les performances du processus le plus lent.

Nous avons introduit dans cette section l'évolution et les concepts liés au calcul haute performance, nous allons maintenant dresser un tour d'horizon de la virtualisation.

1.3 La virtualisation

Dans cette section, on se propose de montrer le fonctionnement et l'intérêt de la virtualisation.

1.3.1 Historique et définition

La virtualisation apparait dès les années 1960 avec un premier système de virtualisation de serveurs réalisés par IBM. À cette époque, l'informatique est peu présente et seules quelques entreprises sont équipées de gros calculateurs, les mainframes. Les mainframes sont des ordinateurs centralisés de grande puissance chargée de gérer les sessions d'utilisateurs des différents terminaux qui lui sont reliés. Tous les traitements sont réalisés par l'ordinateur central. L'utilisation de mainframes présente déjà des soucis d'optimisation de leurs ressources matérielles, ces supercalculateurs étant parfois sous-utilisés. Pour répondre à cette problématique, un premier système est introduit en 1967 par le centre de recherche d'IBM de Cambridge sur l'ordinateur 360/67. Ce système nommé CP/CMS est basé sur un générateur de machines virtuelles (CP) et un logiciel interactif d'interprétation de commandes (CMS) est considéré comme le premier système de virtualisation de serveurs.

Les années 1980-1990 voient apparaitre l'architecture x86 et la commercialisation à large échelle des ordinateurs personnels (PC). L'informatique se fait alors plus présente dans les entreprises et le calcul n'est plus restreint à un ordinateur centralisé. Pour autant, certains traitements nécessitent d'être réalisés de manière partagée comme c'est le cas pour l'accès à une base de données et doivent être accessibles à un ensemble d'utilisateurs. Ce constat entraine l'émergence de l'informatique distribuée, c'est-à-dire l'introduction d'applications "client-serveur". Le développement par Microsoft et Linux de ces applications sur plateforme x86 a fait des serveurs x86 une norme pour l'industrie. Cette normalisation entraine de nouveaux défis aussi bien en terme d'exploitation que d'infrastructure des systèmes distribués. Parmi les défis on peut citer :

- Faible utilisation de l'infrastructure ;
- Augmentation des coûts de l'infrastructure physique ;
- Augmentation des coûts de gestion des postes de travail ;
- Utilisation à haute maintenance préventive et curative ;

La faible utilisation de l'infrastructure et l'augmentation des coûts de l'infrastructure vont devenir deux problématiques importantes du fait de l'introduction du processeur multicœur. De nombreuses applications étant de nature séquentielle, elles n'exploitent qu'un cœur par processeur entrainant de facto une faible utilisation des ressources. Il est alors nécessaire de partager et mutualiser des ressources de calcul entre plusieurs applications. Cette problématique recoupe celle de l'introduction des machines virtuelles dans les années 1960 entrainant ainsi le retour en force de la virtualisation. Pour répondre au défi des serveurs x86 et en faire des plates-formes partagées, VMWare introduit dans les années 2000 le concept de virtualisation totale du matériel x86.

1.3.2 Intérêt de la virtualisation

Nous avons ciblé ici les apports de la virtualisation, particulièrement adaptée aux besoins du calcul haute performance que sont : la conception d'environnement de programmation stable et la gestion dynamique des tâches s'exécutant sur un supercalculateur.

1.3.2.1 Environnement prêt à l'emploi et codes patrimoniaux

Lorsqu'un algorithme est implémenté sous forme de programme, le programmeur s'appuie sur une base d'outils indispensables tels qu'un compilateur et des bibliothèques de programmation. Sans ces outils, le programmeur serait obligé de réinventer un système entier à chaque nouveau programme. On appelle environnement de programmation, l'ensemble des acteurs permettant la traduction en langage machine et l'exécution du programme. Une partie du fonctionnement d'un programme dépend de l'environnement dans lequel il s'exécute et échappe au programmateur, ce qui peut être problématique. En effet, la validation d'un code se fait donc sur la base de son comportement au sein d'un environnement spécifique. L'adaptation d'un code à un nouvel environnement demande donc de le réécrire en partie, mais aussi de le valider. Cela entraine une perte financière et de temps potentiellement très important.

Les codes patrimoniaux sont un exemple direct de cette problématique, ces codes ont été écrits et validés au sein d'environnements différents de ceux présents sur les nouvelles architectures de machine. Il est donc nécessaire de s'attaquer à ce problème de pérennité. La problématique ne se résume pour autant pas à ces anciens codes, en effet les cycles de développement de codes scientifiques sont désormais plus longs que le cycle de vie d'une machine et le code doit être adapté aux nouvelles machines ainsi qu'aux changements de bibliothèques et de supports applicatifs. Pour répondre à ce problème, l'utilisation de machines virtuelles peut être une première source de pérennisation d'un environnement de programmation, mais aussi elle facilite le déploiement d'un code sur une nouvelle machine.

La virtualisation permet aux environnements de programmation d'être moins dépendants d'une architecture de machine particulière. Grâce à la virtualisation, ce n'est plus le code qui s'adapte à la machine, mais c'est la machine virtuelle qui s'adapte à la nouvelle architecture matérielle. L'application scientifique développée n'a plus à être modifiée à chaque changement d'environnement comme c'est le cas aujourd'hui.

1.3.2.2 Répartition dynamique de charge et migration

La gestion des ressources de calcul d'un cluster est aujourd'hui réalisée de manière statique, c'est-à-dire qu'une demande de ressources ne peut être réalisée en cours d'exécution d'un programme. Lors de l'exécution d'une application sur un supercalculateur. L'utilisateur précise le nombre de ressources qu'il souhaite utiliser pour l'intégralité de son exécution aux gestionnaires de ressources.

Cette contrainte de réservation statique entraine une fragmentation de la machine, c'est-à-dire la présence de petits groupes de ressources isolées au milieu de ressources voisines réservées. La fragmentation entraine une sous-utilisation de ces ressources disponibles isolées, en effet elles seront plus difficiles à attribuer du fait de leurs caractéristiques non extensibles. De plus lorsqu'elles sont attribuées, le regroupement de ces ressources isolées entraine une perte en localité au sein du supercalculateur pour les processus de l'application qui en a obtenu l'accès. Les performances de l'application, et

plus particulièrement les performances des communications réseau, peuvent se retrouver altérées sans amélioration possible pendant l'exécution du programme. Lorsque d'autres processus sont libérés, la migration de processus, peut permettre d'améliorer ce genre de situation. Les machines virtuelles intègrent des mécanismes de migration plus performants et robustes que ceux des processus Unix et seraient une solution intéressante pour mettre en place une gestion dynamique d'un supercalculateur.

La migration peut aussi jouer un rôle dans la tolérance aux pannes en permettant de déplacer un job dont les dispositifs de surveillance détectent un comportement laissant présager une panne future. La migration peut être couplée à des mécanismes de copie/-reprise efficace et optimisée pour assurer qu'une simulation ne soit pas intégralement perdue en cas de panne. Dans les années futures, la tolérance aux pannes va prendre une place importante dans les services d'un supercalculateur. En effet, les composants d'un supercalculateur ne peuvent avoir une probabilité de panne nulle par conséquent l'augmentation du nombre de composants entraine mathématiquement une augmentation de la probabilité d'une panne. On ne parle pas ici d'une panne globale qui peut-être prévenue par la redondance des infrastructures essentielles au fonctionnement d'un supercalculateur, mais de panne matérielle locale comme une mémoire ou un périphérique défaillants. Les machines virtuelles ne sont pas le seul moyen de réaliser de la tolérance aux pannes, mais elles ont l'avantage d'avoir été conçues pour pallier ce genre de problématiques.

1.3.3 Famille de solution de virtualisation

Les solutions de virtualisation sont nombreuses et présentent des caractéristiques diverses. Nous parlerons brièvement de l'isolation qui n'est pas une réelle technique de virtualisation puis nous détaillons les trois familles de virtualisation du matériel que sont : la paravirtualisation, la virtualisation complète et la virtualisation assistée par le matériel. On s'intéresse dans notre cas tout particulièrement à l'application de ces trois familles aux CPU x86.

1.3.3.1 Virtualisation par isolation

La virtualisation par isolation consiste à gérer des contextes dans lesquels les processus auront accès à un ensemble restreint de ressources qu'elles soient de type matériel ou logiciel. Dans un système UNIX par exemple, on peut utiliser la commande *chroot* pour isoler un processus dans une sous-arborescence du système de fichier. On peut aussi décider de restreindre l'accès à des ressources comme la mémoire ou les CPU par le biais de *cgroups*. Dans tous les cas, il existe une zone dite principale capable d'accéder à tout le système. L'isolation est une solution simple techniquement et peu consommatrice en terme de surcoût lié à la virtualisation. Les ressources matérielles telles que les disques ou cartes réseau sont facilement partagées entre la zone principale et les *containers* qu'elle contient. Parmi les solutions pour système d'exploitation UNIX, on trouve un grand nombre de solutions telles que LXC, Linux-VServer, BSD Jails, et les Zones Solaris. L'inconvénient majeur est l'impossibilité de virtualiser des systèmes d'exploitation différents du système d'exploitation principal.

1.3.3.2 Virtualisation du matériel

La virtualisation du matériel consiste à faire fonctionner un système d'exploitation comme un simple logiciel. Ce procédé permet à plusieurs systèmes d'exploitation d'utiliser une même machine physique par le biais d'un hyperviseur. Un Moniteur de machine virtuelle (VMM) ou hyperviseur [RG05] peut s'exécuter directement sur le matériel ou

de manière logicielle au-dessus du système d'exploitation d'une machine physique. Cette caractéristique ajoute un niveau d'abstraction permettant de fournir des environnements variés sans modifier la configuration d'une machine, facilitant ainsi l'administration d'un cluster.

Para virtualisation La para virtualisation permet à une machine virtuelle d'utiliser des ressources physiques. L'architecture virtuelle exposée est légèrement différente de l'architecture physique sous-jacente. Cette différence empêche la compatibilité entre les pilotes des systèmes d'exploitation et la virtualisation de l'architecture physique. Il est donc nécessaire de modifier le code du système d'exploitation pour rétablir cette compatibilité, ce désavantage est le principal handicap de la para virtualisation. Pour autant, la para virtualisation permet une plus grande liberté dans l'implémentation des interactions entre le système d'exploitation et l'architecture virtuelle. La para virtualisation est utilisée depuis longtemps dans des hyperviseurs tels que VM/370 [Cre81] et Disco [BDGR97]. Pour cela, ils utilisaient une combinaison d'instructions, de registres et de périphériques vers des architectures virtuelles afin d'en améliorer les performances. Ces systèmes avaient pour objectif d'exécuter des systèmes d'exploitation inadaptés aux nouvelles architectures.

Virtualisation complète La virtualisation est dite complète lorsque le système d'exploitation invité n'a pas conscience d'être virtualisé. L'OS qui est virtualisé n'a aucun moyen de savoir qu'il partage le matériel avec d'autres OS. Ainsi, l'ensemble des systèmes d'exploitation virtualisés s'exécutant sur un unique ordinateur, peuvent fonctionner de manière totalement indépendante les uns des autres et être vus comme des ordinateurs à part entière sur un réseau.

Virtualisation assistée par support matériel Les extensions matérielles pour machine virtuelle sont fournies sur les processeurs Intel (Intel Virtualization Technology (Intel VT)[UNR+05]) et AMD (AMD-V [AMD08]). Ces extensions permettent de supporter deux types de logiciels : l'hyperviseur qui se comporte comme un hôte réel et qui a le contrôle complet du processeur et des autres parties matérielles et le système invité, qui est exécuté dans une machine virtuelle (VM). Chacune des machines virtuelles s'exécute indépendamment des autres et utilise la même interface pour le processeur, la mémoire et autres périphériques fournis par la plateforme physique. Ces extensions améliorent les performances des applications lancées au sein d'une machine virtuelle.

1.3.4 Hyperviseurs et outils de virtualisation

Les solutions propriétaires ou open source de virtualisation sont nombreuses. Parmi celles-ci on trouve deux solutions open source : Xen [BDF+03] et QEMU-KVM [Bel05] que nous décrivons dans la suite de cette section. En parallèle des hyperviseurs classiques, on trouve des solutions de gestion de cluster virtuel, comme la solution open source LibVirt.

1.3.4.1 Xen

Xen est un hyperviseur open source « bare metal », aussi appelé hyperviseur de « type 1 ». Le code de Xen est exécuté juste au-dessus du matériel, avec un niveau de privilèges d'accès aux ressources système (CPU, IO, RAM) maximal. Xen implémente certaines fonctions qu'on retrouve dans tous les OS : un ordonnanceur, un gestionnaire d'interruptions, quelques drivers, etc. Les facilités d'administration de l'hyperviseur sont accessibles à travers une machine virtuelle spécialement privilégiée et instanciée tout de

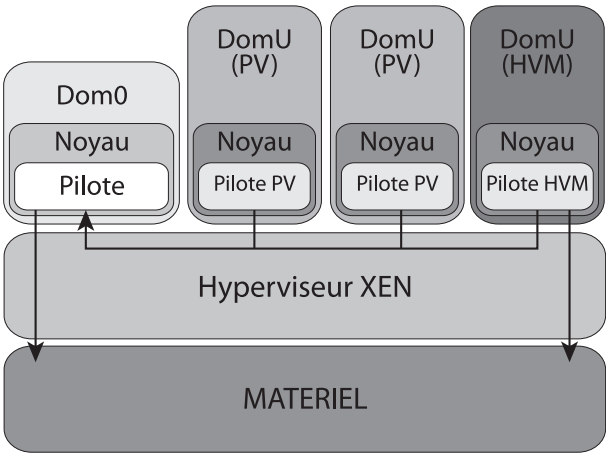


FIGURE 1.14 – Machines virtuelles et hyperviseur XEN

suite après le démarrage de l'hyperviseur. Cette machine virtuelle appelée « domain 0 » est un système d'exploitation complet (au choix Linux, BSD, Solaris, etc.) dont le noyau est modifié pour communiquer avec l'hyperviseur sous-jacent. Le dom0 contrôle (démarre, arrête, surveille) les autres machines virtuelles non privilégiées (appelées « User domain » ou domU).

Les domU peuvent être paravirtualisés ou matériellement virtualisés. La paravirtualisation implique une modification du système d'exploitation invité afin de coopérer avec l'hyperviseur pour l'accès aux ressources physiques. Les opérations d'entrée/sortie sont principalement réalisées par le dom0 qui effectue les opérations d'E/S à la place de la machine virtuelle. La virtualisation matérielle ne nécessite pas de modifications de l'OS invité, car le CPU, au travers d'un jeu d'instructions spéciales (Intel VT-d ou AMD-V), fait croire au système invité qu'il a un accès direct au matériel. Cependant, cela entraîne un surcoût de fonctionnement, car certaines demandes d'accès aux ressources physiques effectués par la VM doivent être interceptées pour vérifier qu'elles n'affectent pas les autres VM.

1.3.4.2 QEMU-KVM

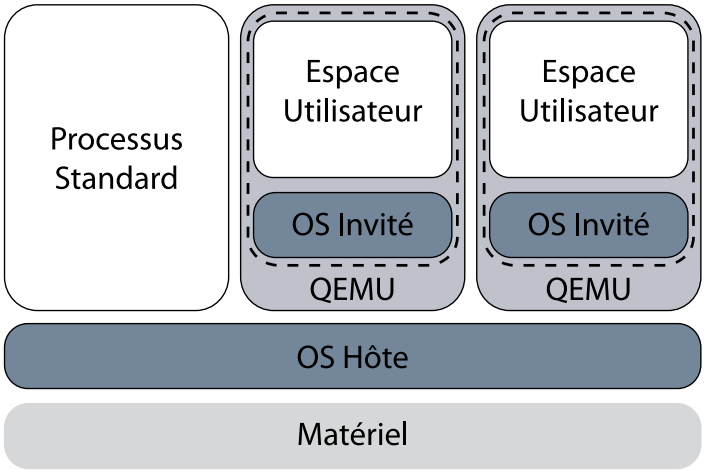


FIGURE 1.15 – Machines virtuelles et hyperviseur QEMU-KVM

Quick EMUlator (QEMU) QEMU émule les ressources nécessaires à une machine virtuelle. Parmi ces émulations, on trouve celle des processeurs, les Virtual Central Processing Unit (VCPU), mais aussi un ensemble de périphériques tel que des cartes graphiques ou des cartes réseau. Pour le système hôte, QEMU est vu comme un processus UNIX standard et les VCPU sont des threads. C'est ce processus UNIX qui contient le système invité. Lorsque QEMU est utilisé seul, les VCPU sont complètement émulés. La machine virtuelle est alors totalement isolée du système hôte. L'utilisation de QEMU avec émulation complète d'un processeur est coûteuse et peut dégrader fortement les performances (d'un facteur 5 à 10). Pour avoir des performances proches d'un système natif, il faut utiliser un module d'accélération.

Kernel based Virtual Machine (KVM) KVM est un module noyau intégré au noyau LINUX depuis 2007 qui permet à ce dernier de remplir le rôle d'un hyperviseur. KVM supporte un grand nombre d'architectures telles que les architectures x86 disposant des instructions Intel VT ou AMD-V, mais aussi Power PC, IA-64 ou encore ARM. Il permet l'utilisation d'extensions matérielles (HVM) telles que des jeux d'instructions dédiés à la virtualisation. Parmi celles-ci figure l'exécution d'un coeur en mode invité. Ce mode est orthogonal aux deux modes classiques que sont le mode utilisateur et le mode noyau. En mode invité, un processeur a accès à tous les niveaux de privilèges du x86 de 0 à 3. Les instructions de la machine virtuelle invitée sont alors directement exécutées sur le processeur. Une partie des instructions privilégiées déclenchent des interruptions ce qui permet à l'hyperviseur d'intervenir quand il le faut. Grâce à KVM un thread VCPU utilise le mode invité. Le coeur sur lequel s'exécute un VCPU peut alors par le principe du trap-and-emulate diminuer ainsi le coût de la virtualisation. Pour autant les VCPU sont des threads normaux du point de vue de l'ordonnanceur LINUX.

1.3.4.3 Libvirt

Libvirt est un ensemble de logiciels open source qui permet d'interagir avec de multiples technologies de virtualisation. Libvirt fournit une API en langage C, un démon appelé libvirtd ainsi qu'un utilitaire en ligne de commande, virsh. Libvirt est un projet Red Hat développé activement depuis 2005. De nombreuses technologies de virtualisation sont supportées, parmi lesquelles VirtualBox, OpenVZ, KVM/QEMU, VMware Workstation, Xen, LXC... L'API expose en particulier des fonctions permettant de contrôler des machines virtuelles.

1.4 Virtualisation et Mémoire sur architecture X86

1.4.1 Mémoire paginée : Cas standard

La gestion mémoire, et plus précisément la pagination, impacte directement les performances globales d'un programme. Nous nous sommes donc intéressés tout particulièrement à cet aspect. L'objectif de ce chapitre est de montrer le fonctionnement standard d'une table des pages dans le cas natif et dans le cas virtualisé.

1.4.1.1 Table des pages

Les pages standards correspondent à des portions de mémoire de taille fixée à 4 Ko. Ainsi, 12 582 912 pages sont nécessaires pour adresser une mémoire de 48 Go. Pour des raisons d'efficacité mémoire, il n'est pas envisageable de stocker un index détenant autant d'entrées. Afin d'y remédier, la table des pages est hiérarchisée : l'espace occupé

par les pages constituant la table des pages est ainsi réduit. En outre, chaque niveau d'index occupe une place mémoire équivalente à 4 Ko.

Memory manager unit L'utilisation d'une table des pages hiérarchique pour la traduction d'adresses virtuelles en adresses physiques génère un surcoût lors de l'accès à une donnée. Afin qu'elle soit faite de manière efficace, cette traduction ou pagewalk est réalisée par une MMU. Cette dernière est gérée de manière matérielle par le processeur. C'est pour cette raison que la pagination est normée et peu flexible. Nous présentons dans cette synthèse la pagination mémoire sur les architectures x86 pour les systèmes 32 et 64 bits. Dans les chapitres suivants, nous ciblerons plus particulièrement les systèmes de type 64 bits (x86_64).

Système 32 bits, x86 Dans un système UNIX standard avec des adresses sur 32 bits, la table des pages possède deux niveaux d'index qui permettent d'accéder aux pages de 4 Ko. Les adresses étant codées sur 32 bits, une page standard peut contenir jusqu'à 2^{10} adresses. Cette structure en arbre est parcourue pour obtenir l'adresse physique d'une donnée à l'aide de son adresse virtuelle. Ce mécanisme de traduction aussi nommé pagewalk se déroule sur un système 32 bits comme suit. Tout d'abord, on lit l'adresse de l'index racine qui est stockée dans un registre du processeur, le CR3. Puis les 10 premiers bits de poids fort de l'adresse virtuelle à traduire désignent l'emplacement de l'adresse de la page contenant le second niveau d'indirection. De la même manière, on accède à l'adresse pointant vers le prochain niveau d'index et on obtient l'adresse de la page de 4 Ko recherchée. Les 12 bits restants donnent le décalage (*offset*), depuis le début de la page, à réaliser pour accéder à la donnée souhaitée.

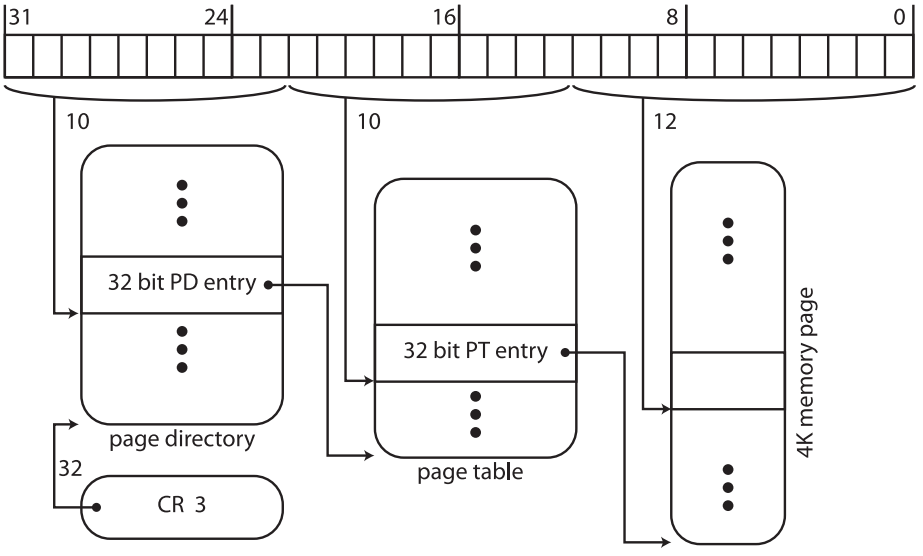


FIGURE 1.16 – Hiérarchie d'une table des pages (système x86 32 bits)

Système 64 bits, x86_64 Dans un système x86_64, les adresses sont codées sur 48 bits et non 64 comme on pourrait le penser. Cela permet tout de même d'adresser 256 To de données ce qui est largement suffisant pour les systèmes actuels. La table des pages compte dans ce cas quatre niveaux d'index pour accéder aux pages de 4 Ko. L'index utilise deux fois plus d'espace mémoire, une page ne contient plus que 2^9 adresses. Le mécanisme de pagewalk est similaire aux systèmes 32 bits (x86), mais à présent, neuf bits sont utilisés pour retrouver l'adresse du niveau suivant. Les bits utilisés pour parcourir les quatre niveaux sont, en partant de l'index racine : 39-47, 30-38, 21-29 et 12-20. Les 12 bits restants permettent de se placer au bon endroit dans la page recherchée.

Avantage des pages de 4 Ko Le choix d'une taille de 4 Ko est issu d'un compromis entre la complexité du mécanisme de parcours de la table des pages et l'utilisation effective de la mémoire (Mémoire utilisée par l'application / Mémoire allouée par le système). En effet, plus la taille des pages est petite plus on pourra approximer fidèlement la quantité de mémoire demandée par un utilisateur. Un autre avantage de cette solution réside dans le faible coût de la remise à zéro des pages. Toutefois, il n'est pas possible de choisir une taille de page trop petite. Cela entrainerait plus de niveaux d'indirections et donc un parcours de la table des pages plus coûteux. De plus, une allocation de très grande taille nécessiterait un nombre important de pages.

1.4.1.2 Translation lookaside buffer (TLB)

Par souci d'efficacité, la MMU des processeurs actuels est capable de réaliser le mécanisme de pagewalk de manière matérielle. Le parcours restant coûteux, elle possède un cache appelé la TLB.

1.4.1.3 MMU virtuelle logicielle

KVM permet aussi d'émuler une MMU de manière logicielle à l'aide d'une table des pages fantôme (shadowpage), cette émulation est utilisée par défaut si le processeur ne possède pas les extensions nécessaires aux MMU virtuelles matérielles. La table des pages fantôme est la table des pages utilisée par la VM. Elle est répliquée par l'hyperviseur à partir de la table des pages de l'hôte. Après conversion, la nouvelle table fournit alors la translation page machine virtuelle vers page machine physique. L'unité de gestion de la mémoire pointe alors vers la table des pages fantôme pendant l'exécution de la machine virtuelle (voir figure 1.17).

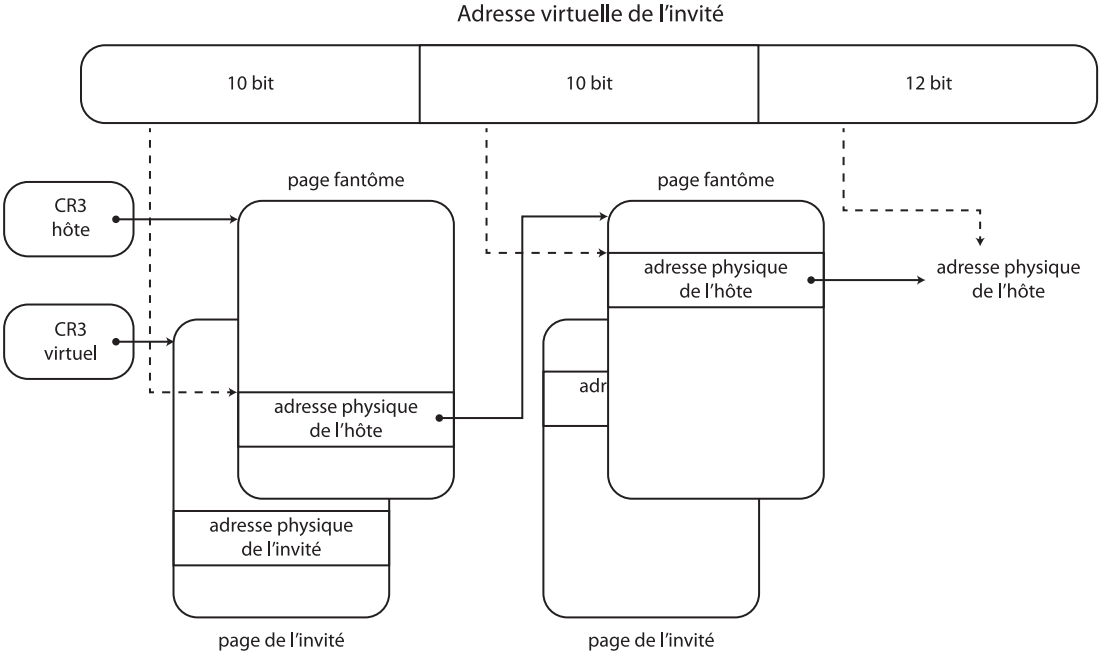


FIGURE 1.17 – Mécanisme de la table des pages fantôme

Le maintien de la cohérence entre table des pages invitée et table des pages fantôme constitue la principale difficulté de ce mécanisme. Le surcoût de cette cohérence pouvant être important dans le cas d'une utilisation très dynamique de la table des pages.

1.4.1.4 MMU virtuelle matérielle

L'emploi d'une MMU virtuelle matérielle se fait par l'utilisation d'extensions ajoutées sur les processeurs récents. Ce concept a d'abord été implémenté par AMD sous le nom de table des pages imbriquée (*Nested-Page*). Il est aussi disponible sur les processeurs INTEL sous la forme d'une table des pages étendue (EPT).

La table des pages imbriquée permet à chaque machine virtuelle (VM) de disposer d'une MMU dédiée. Cette MMU contient deux tables distinctes, l'une étant stockée par l'hyperviseur et l'autre par le système invité. La table des pages remplie par l'hyperviseur contient une traduction des adresses virtuelles de l'invité vers les adresses physiques de l'hôte correspondante. Le système invité quant à lui installe sa propre table de pages avec une traduction des adresses virtuelles invitées vers les adresses virtuelles de l'hôte. L'utilisation conjointe des deux tables permet la traduction complète de l'adresse d'une page virtuelle à celle d'une adresse de page machine physique (figure 1.18)

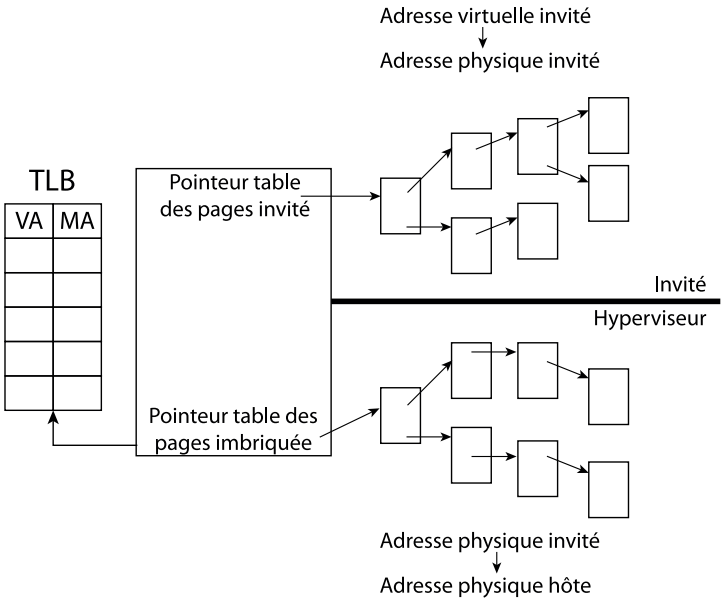


FIGURE 1.18 – Mécanisme de la table des pages étendue (EPT)

Translation Lookaside Buffer Miss Pour chaque niveau de la table des pages de l'invité, il faut traduire les adresses machines virtuelles de l'invité pour obtenir les adresses physiques hôtes de la page. Comme le montre la figure 1.19, il faut commencer par se placer sur l'index racine de l'invité. Pour cela, de la même manière que pour une traduction native, on commence par lire l'adresse contenue dans le registre CR3 de l'invité. Cette adresse est, du point de vue de l'hôte, une adresse virtuelle. Elle est donc traduite par le pagewalk de la table des pages stockée par l'hyperviseur. Il faut alors parcourir les quatre niveaux de la table des pages imbriquée.

Sur la figure 1.19, les adresses nL représentent les accès liés aux pagewalk permettant la traduction des adresses physiques invitées vers les adresses physiques hôtes. Les adresses gL sont les adresses virtuelles de la table des pages dédiée à l'invité ainsi que l'adresse finale recherchée, notée *SPA*. On peut alors calculer le coût d'un TLB miss qui demande un pagewalk de 24 accès mémoire, c'est le principal handicap d'une MMU virtuelle de type matériel.

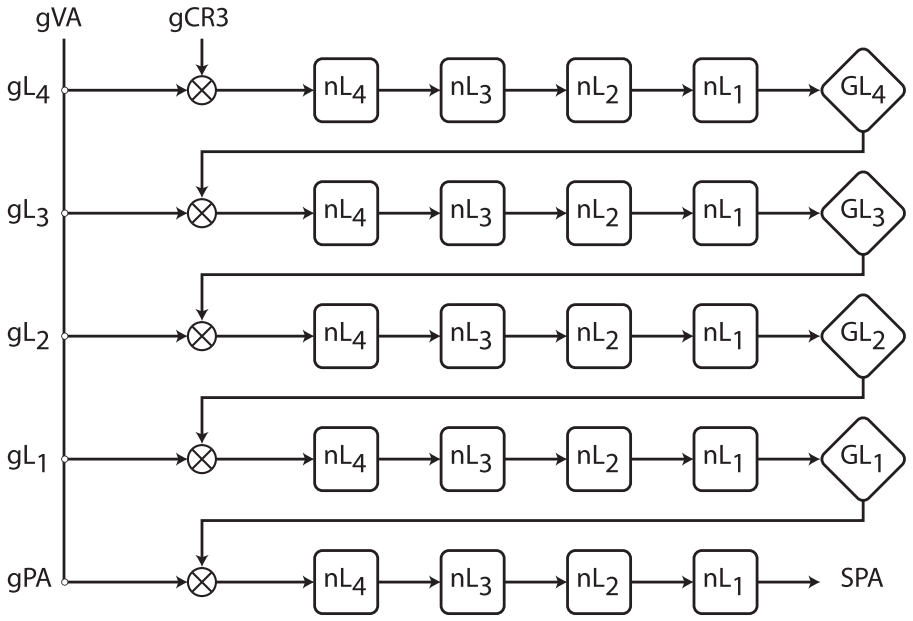


FIGURE 1.19 – Parcours d'une table des pages imbriquée (*Nested-Page*)

1.4.2 Mémoire paginée : Cas des grosses pages

Dans cette partie, nous présentons deux nouveaux types de pages ainsi qu'un algorithme permettant une meilleure utilisation de la mémoire lors d'allocations de grande taille comme celle de la RAM d'une machine virtuelle.

1.4.2.1 Grosses pages et table des pages

La taille de 4 Ko n'est pas toujours la plus adaptée, notamment lors de l'utilisation de grosses allocations de l'ordre du gigaoctet ou du téraoctet. Les systèmes et processeurs actuels permettent d'utiliser des pages de taille plus grande appelées grosses pages. Pour un système x86_64, les tailles disponibles sont 2 Mo et 1 Go. Les pages de 2 Mo permettent de se ramener à une table des pages à trois niveaux alors que les pages de 1 Go n'utilisent plus que deux niveaux. Ces tailles sont calculées pour permettre de sauter exactement un niveau de table des pages. Ainsi pour un système 32 bits les grosses pages ont une taille de 4 Mo.

Le recours à des grosses pages permet de manière diminuer directement le coût du pagewalk. Les grosses pages peuvent être utilisées de manière transparente mais peuvent entrainer en pratique une surconsommation de la mémoire. En effet, les techniques d'allocations ne sont pas les mêmes pour les grosses pages et pour les petites pages. Ces modifications s'accompagnent d'un besoin d'optimisation pour pouvoir tirer parti des grosses pages. En outre, la remise à zéro des grosses pages peut, dans le cas de schémas d'allocation mémoire particuliers, entraîner des pertes de performance. Cette remise à zéro des pages est un mécanisme de sécurité qui assure qu'un processus ne puisse pas lire les informations d'une page libérée par un autre processus.

Dans notre cas, c'est l'hyperviseur qui utilise les grosses pages pour allouer la RAM d'une machine virtuelle. L'hyperviseur QEMU-KVM peut fonctionner avec des grosses pages sans besoin de modification. Les benchmarks que nous effectuons sont réalisés dans le système invité et utilisent quant à eux des petites pages.

1.4.2.2 Grosses pages et TLB

Grâce aux grosses pages, le coût d'un TLB *miss* va diminuer. Le pagewalk se fait avec un ou deux niveaux en moins. Précisons que les entrées de la TLB contenant les traductions des adresses pointant vers les grosses pages sont mémorisées dans un tableau indépendant de celui des pages standards. Ce cache est deux fois plus petit que celui des pages standards. Toutefois il permet d'indexer 256 fois plus de données (celles-ci étant plus nombreuses par page). Le risque d'évincement d'une entrée de la TLB est donc réduit si l'on utilise des grosses pages et un espace mémoire important. Ainsi, quand le nombre de pages standards demandées par un utilisateur est supérieur à la taille de la TLB, il devient encore plus intéressant d'utiliser des grosses pages.

Dans la section précédente, nous avons montré le pagewalk en 24 accès inhérent aux MMU virtuelles imbriquées. L'utilisation des grosses pages dans l'hôte permet de gagner un niveau sur la traduction des adresses physiques du système invité vers les adresses physiques hôtes. Le nombre d'étapes passe à dix-neuf lorsque l'hyperviseur manipule des grosses pages de 2 Mo et à quatorze lors de l'utilisation de grosses pages de 1 Go. Le coût d'un pagewalk d'une MMU matérielle peut donc être divisé par deux. Nos benchmarks utilisent essentiellement des petites pages, nous ne nous sommes donc pas intéressés à l'utilisation des grosses pages au sein de l'invité, mais uniquement au sein de l'hôte. Nous pourrions encore diminuer le coût d'un parcours de table des pages par l'utilisation conjointe des grosses pages côté hôte et côté invité.

1.4.2.3 Grosses pages transparentes (THP)

Les contraintes liées à l'utilisation des grosses pages sont aussi bien du côté utilisateur qui doit adapter son code, que du côté administrateur. En outre, il peut être compliqué pour un utilisateur de savoir quel type de page est le plus adapté à son application.

L'utilisation des grosses pages passe par l'utilisation d'un système de fichiers particulier (`hugetlbfs`). Ces grosses pages peuvent être réservées à l'aide de droit super-utilisateur ou obtenues à la demande. La réservation des grosses pages rend indisponible l'espace mémoire correspondant pour les pages standards. On ne bénéficie alors plus de la mémoire complète si toutes les pages utilisées sont standards. A contrario, le système peut ne pas être capable de fournir de grosses pages à l'utilisateur lorsqu'il les sollicite à l'exécution.

Afin de masquer la complexité de la gestion et de l'utilisation des grosses pages, LINUX intègre un algorithme qui permet au noyau de choisir le type de page qu'il utilise lors de l'exécution d'une application. Cet algorithme est basé sur l'utilisation de grosses pages transparentes (*Transparent Huge Page* - THP). L'objectif affiché de cet algorithme est de rendre totalement transparente l'utilisation de ces grosses pages.

Nous avons dans un premier temps utilisé les grosses pages transparentes qui sont activées par défaut sur notre plateforme d'expérimentation. En effet, l'algorithme se base sur des heuristiques et hérite de l'historique de la machine. Nous l'avons donc désactivé pour obtenir des performances reproductibles. Nous parlons ici d'historique en terme de mémoire, en ce sens qu'un nœud de calcul présente une mémoire plus ou moins fragmentée de par son utilisation antérieure. De la même manière que pour l'utilisation de grosses pages à la demande, le système peut ne pas être capable de fournir à l'utilisateur des grosses pages transparentes. L'utilisateur ne sait donc pas à priori le type de page que le système utilisera. Cela pose un problème de reproductibilité des résultats.

1.4.2.4 Étude comparative entre table des pages étendue et fantôme

Afin d'illustrer l'importance du choix de la table des pages pour une machine virtuelle, nous avons mesuré l'impact en temps d'exécution sur quelques benchmarks. Notre test consiste à quantifier l'impact en temps d'exécution sur le benchmark BT de la suite NAS-OpenMP et sur une application réelle, Hera qui est une application de type AMR (Adaptative Mesh Refinement) développée par le CEA.

La différence significative entre les deux benchmarks choisis réside dans leur utilisation de la mémoire. En effet, les NAS-OpenMP benchmarks génèrent la mémoire de façon statique. Elle est allouée au début de l'application et libérée à la fin. On n'a donc aucun besoin de contrôler la cohérence et la stabilité de la table des pages. À l'inverse, le raffinement de maillage de l'application Hera implique de nombreuses allocations et désallocations pendant l'exécution du programme. Ce fonctionnement implique un comportement très dynamique des tables de pages invité et hôte.

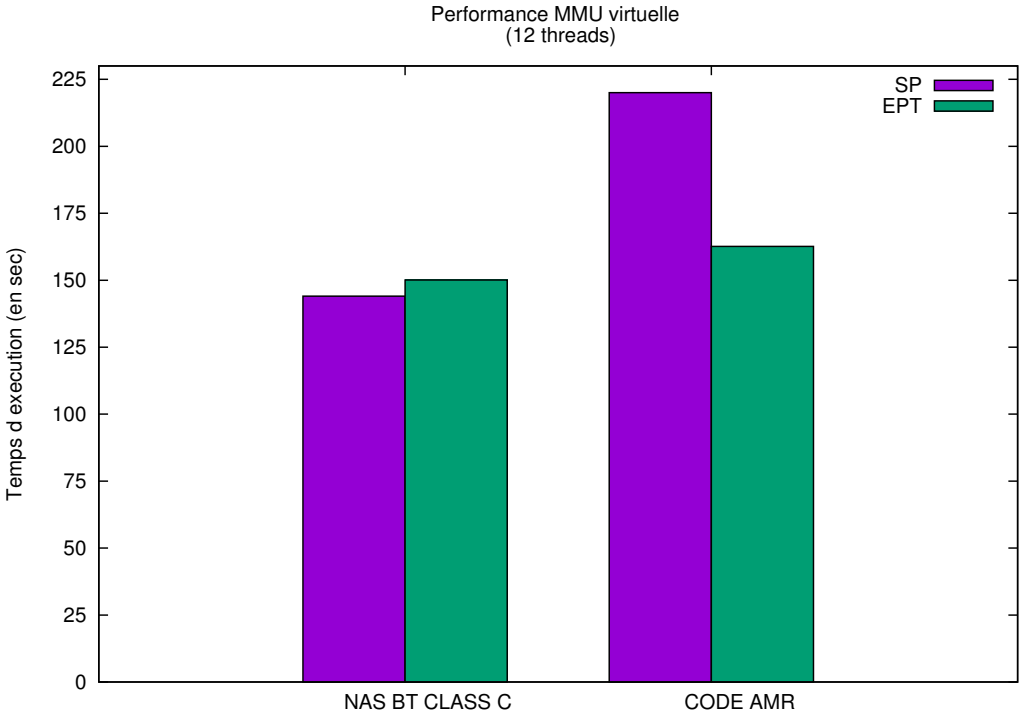


FIGURE 1.20 – Comparaison entre EPT et table des pages fantôme

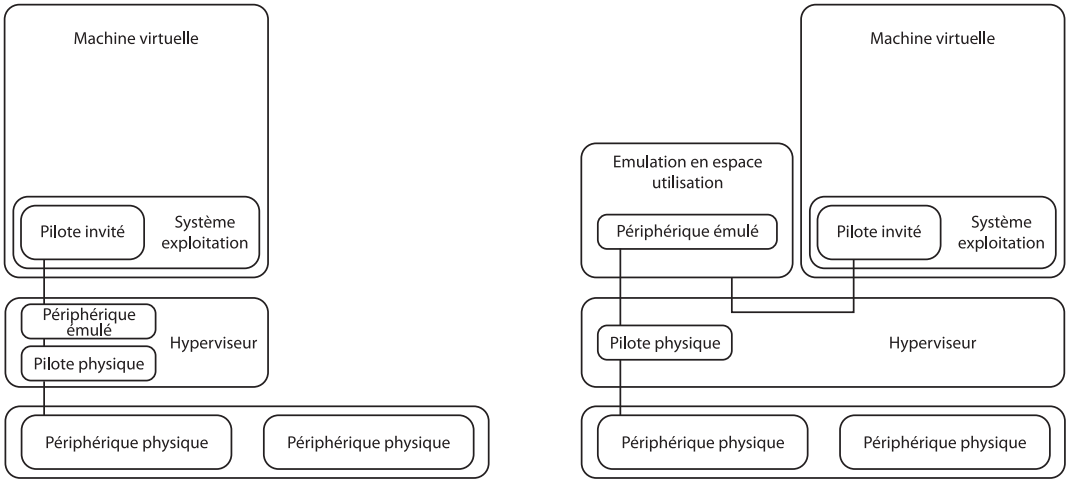
Comme nous pouvons le constater sur la figure 1.20, le NAS BT a un temps d'exécution plus faible avec une MMU logicielle. Cela vient confirmer l'hypothèse selon laquelle une MMU virtuelle logicielle est plus performante qu'une MMU virtuelle matérielle lorsqu'il n'y a pas de cohérence à gérer. Il est à noter que l'écart n'est pas significatif dans le cas des NAS. Par contre dans le cas d'un code AMR, le choix d'une MMU matérielle s'impose. Une bonne connaissance de l'application est donc nécessaire pour en déduire le mode d'exécution le plus favorable dans un environnement virtualisé.

1.5 Virtualisation et Communications réseau

Les codes utilisant le modèle à mémoire partagée ne sont pas les seuls codes de calculs que nous regardons. Nous nous intéressons aussi au modèle à mémoire distribuée utilisé

et qui nécessite l'utilisation du réseau d'un cluster. Pour cela, nous avons étudié les différentes techniques permettant les échanges entre plusieurs machines virtuelles.

1.5.1 Émulation d'un périphérique physique



(a) Émulation par l'hyperviseur (b) Émulation en espace utilisateur

FIGURE 1.21 – Emulation d'un périphérique physique

La machine virtuelle, à la différence d'un système d'exploitation, n'a pas accès directement au matériel, celui-ci étant contrôlé par le système d'exploitation hôte. Pour des raisons de fiabilité et de cohérence, on ne peut pas dans le cas classique accéder directement à un périphérique depuis l'invité. Pour permettre à la machine virtuelle d'utiliser le matériel sous-jacent, on a besoin de lui fournir une abstraction du matériel afin de relayer l'utilisation du matériel au sein de l'hôte. Cette abstraction ou émulation peut être réalisée par le biais de l'hyperviseur ou par le biais d'un programme tiers au sein de l'espace invité de l'hôte (non privilégié) comme le montre les figures 1.21a et 1.21b.

L'émulation ajoute un niveau d'abstraction et présente un surcoût important dans l'accès aux ressources de l'hôte. Pour y remédier des solutions matérielles comme PCI-passthrough ou SR-IOV [DYL+12] que nous présentons dans la suite ont été proposées. De plus, des solutions en espace noyau de l'hôte ont été ajoutées pour limiter le coût de l'abstraction d'un périphérique.

1.5.2 PCI Passthrough

Fonctionnement Le PCI Passthrough permet d'assigner directement à une machine virtuelle un périphérique PCI physique de l'hôte. Le système d'exploitation invité peut alors utiliser le périphérique matériel directement sans en rendre compte au système d'exploitation hôte. Pour cela, on détache le périphérique du pilote du système d'exploitation hôte de manière explicite puis on l'attache à l'invité. La figure 1.22 illustre l'utilisation d'un périphérique en PCI passthrough.

Limites Quand un périphérique PCI est directement assigné à un invité, la migration nécessite que le périphérique soit détaché de l'invité pour être réalisé.

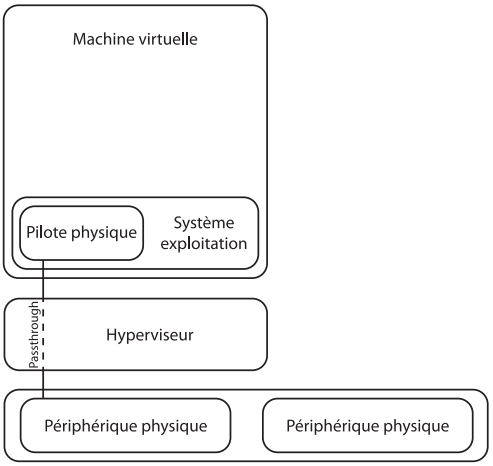


FIGURE 1.22 – PCI passthrough

1.5.3 Single Root I/O Virtualisation

La spécification SR-IOV permet à un périphérique de type PCIe d'être vu comme une multitude de périphériques PCIe physiques distincts. Cette spécification a été développée et est maintenue par PCI SIG afin d'en faire un standard complet permettant une meilleure interopérabilité. Pour cela, SR-IOV introduit les concepts de fonctions physiques dites PF et de fonctions virtuelles, VF. Les fonctions physiques permettent d'avoir accès à l'ensemble des fonctionnalités PCIe alors que les fonctions virtuelles sont plus légères et ne permettent pas de configurer le périphérique. L'utilisation de SR-IOV nécessite un BIOS ainsi qu'un système d'exploitation et hyperviseur ayant le support adéquat.

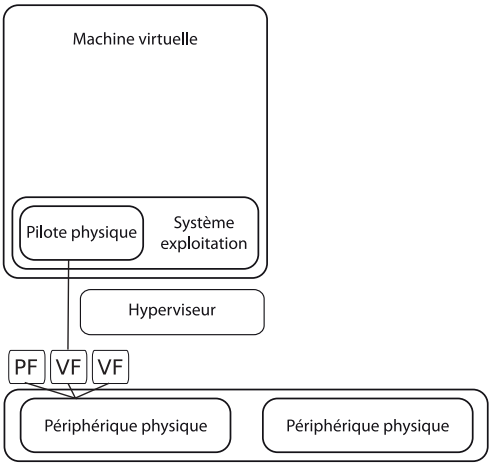


FIGURE 1.23 – Principe du SR-IOV

Fonctionnement Les fonctions physiques sont découvertes et gérées comme n'importe quel périphérique PCIe. Avec les fonctions physiques, il est possible de configurer et contrôler intégralement un périphérique PCIe, les fonctions virtuelles quant à elles permettent d'initialiser le périphérique et de l'utiliser sans impacter les autres processus utilisant le périphérique. L'absence des fonctionnalités de configuration avancée pour les fonctions virtuelles ne permet pas de les substituer de manière transparente aux fonctions physiques. Il est donc nécessaire que le système d'exploitation ainsi que l'hyperviseur sachent s'ils ont accès à un périphérique PCIe complet (natif ou PF) ou limité (VF). L'utilisation de SR-IOV nécessite donc un support spécifique permettant

de détecter et utiliser correctement les fonctions physiques et virtuelles. Aujourd'hui, ce support existe pour la plupart des hyperviseurs open source comme KVM ou Xen, mais aussi dans les hyperviseurs commerciaux comme VMware vSphere ou Microsoft Hyper-V. Comme on l'a vu dans la partie 1.5.2, il est essentiel de pouvoir utiliser des réseaux rapides tels qu'Infiniband[HN] lors des communications réseau d'un code parallèle distribué. SR-IOV permet de lever la contrainte d'un périphérique physique par machine virtuelle. On peut donc utiliser un grand nombre de machines virtuelles par nœud de calcul. La figure 1.23 montre l'utilisation d'un périphérique en SR-IOV, l'hyperviseur comme pour le PCI Passthrough n'intervenant plus dans l'utilisation du périphérique.

Pour autant de la même manière que pour le PCI passthrough, SR-IOV entraine une adhérence plus forte au matériel de l'hôte et complexifie la migration et les mécanismes de copies/reprises.

1.5.4 Périphériques virtuels : interface VirtIO

Nous avons détaillé les solutions matérielles permettant l'accès à un périphérique physique par l'invité. Ces solutions sont contraignantes en terme de matériel et de pilote et ne peuvent être considérées comme une réponse complète au partage de périphérique. Les solutions matérielles ne sont pas les seules alternatives à l'émulation brute, des solutions logicielles ont été proposées pour accélérer l'émulation. Cette accélération est bien souvent basée sur une interaction entre le noyau invité pour alléger le coût d'une action sur un périphérique. Le code du noyau d'un système d'exploitation se doit d'être robuste et ne peut être mise à jour pour chaque solution logicielle. Pour cette raison, KVM intègre l'API Virtio qui permet de partager des files de messages entre un module noyau de l'invité et son hyperviseur. Virtio ,développé pour l'hyperviseur lguest, est maintenant complètement intégré à KVM (figure 1.24).

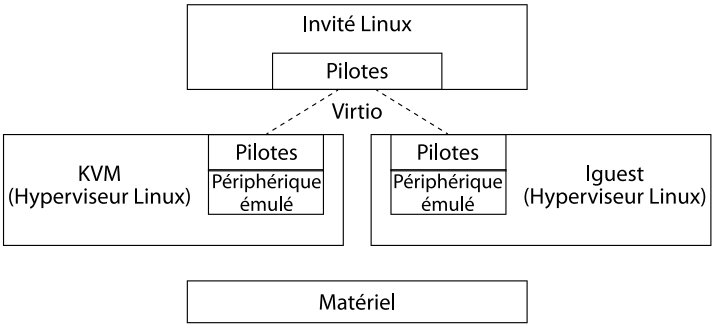


FIGURE 1.24 – Principe de VirtIO

1.5.4.1 Virtio net

Le fonctionnement classique de QEMU consiste à faire émuler les accès entrée/sortie provenant de l'invité par le processus QEMU s'exécutant au sein de l'espace utilisateur de l'hôte. Vhost-net se base sur un périphérique d'émulation virtio inclus dans le noyau Linux. Cela permet de réaliser une émulation appelant directement les fonctions internes du noyau plutôt que de réaliser des appels systèmes depuis l'espace utilisateur. La figure 1.25 illustre le fonctionnement du périphérique virtuel Vhost-net, l'invité communique avec l'espace noyau de l'invité au moyen de buffers organisés en anneaux en écriture et lecture. Il a de plus accès en lecture aux données contenues dans l'invité et des communications par évènements sont réalisées pour faire avancer les messages envoyés et reçus.

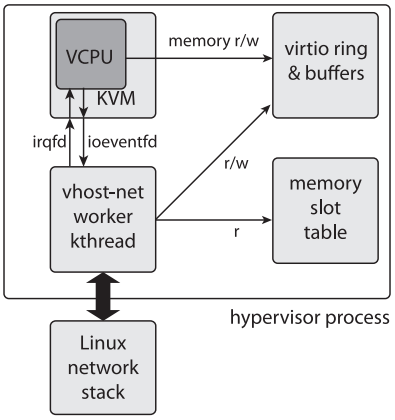


FIGURE 1.25 – Fonctionnement de Vhost-net

1.5.4.2 IVSHMEM

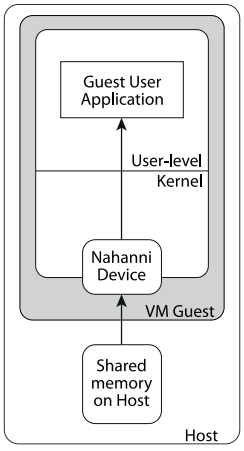


FIGURE 1.26 – Fonctionnement d' IVSHMEM

IVSHMEM est un périphérique virtuel implémenté dans QEMU et qui permet de réaliser un shm_open entre l'hôte et l'invité. Côté hôte, la mémoire est mappée puis tronquée afin de forcer la réservation des pages physiques associées à la zone mémoire mappée. Le descripteur de fichier associé à la zone mémoire ainsi réservée est passé en argument de QEMU pour initialiser un périphérique PCI. Coté invité, la zone mémoire est accessible via l'utilisation d'un module noyau qui fournit le pilote nécessaire à l'initialisation du périphérique PCI. IVSHMEM fournissant une interface robuste et une liste de mainteneurs actifs, il nous a paru intéressant de tirer parti de cet outil.

1.5.4.3 Vcluster

Vcluster est une preuve de concept permettant de déployer un cluster de machine virtuelle monocœur exécutant un même code MPI. Pour cela, Vcluster fournit son propre hyperviseur basé sur KVM ainsi qu'un périphérique virtio dédié aux communications MPI. Ces communications peuvent être de nature interne à un nœud de calcul et exploitent de la mémoire partagée ou entre des nœuds de calcul grâce au protocole MPI ou une implémentation Infiniband de type verbs. Vcluster permet d'obtenir un faible surcoût induit par la virtualisation pour des codes purs MPI, il intègre un mode de déploiement distribué permettant de réduire le temps de démarrage d'un programme

MPI. Pour autant, Vcluster reste une preuve de concept avec un hyperviseur léger permettant un ordonnancement de ses services peu coûteux et capables de fournir tous les points d'entrées nécessaires au pilote vcli. Ces caractéristiques ne peuvent être transposées à des outils de virtualisation complète comme Xen ou QEMU-KVM. Pour ces raisons, l'intégration de Vcluster au sein d'un projet comme QEMU n'est de notre point de vue pas envisageable, mais fournit une base de connaissance propre à développer une solution QEMU intégrant les contraintes du HPC.

L'objectif de la thèse étant de fournir une solution générique dédiée à la virtualisation d'environnement, nous avons choisi de nous baser d'un hyperviseur intégrant tous les services qu'intègre un système d'exploitation classique. Dans notre cas, nous avons choisi QEMU-KVM et suivi une approche reposant sur le mode de fonctionnement d' IVSHMEM.

Chapitre 2

Virtualisation et applications scientifiques multithreadées

"L'art de diriger consiste à savoir abandonner la baguette pour ne pas gêner l'orchestre"

Herbert von Karajan

Nous avons vu que la virtualisation peut être utilisée pour partager les ressources d'une machine physique entre plusieurs machines virtuelles. Dans la consolidation de serveur par exemple différents services peuvent cohabiter au sein d'une même machine physique. Les services s'exécutent dans des machines virtuelles afin de les cloisonner et de leur associer un environnement d'exécution adapté à leur besoin. Le nombre de machines virtuelles et le nombre de ressources dédiées à une machine virtuelle peuvent varier en fonction du nombre d'utilisateurs d'un service ou de la spécificité des traitements inhérents à un service. Cette hétérogénéité temporelle de besoin de ressources est en général moins marquée pour des applications HPC en raison de leur régularité durant leur exécution. Pour autant, les applications HPC peuvent être découpées en phase de calcul spécifique comme une phase lecture ou écriture de données ou l'appel à des solveurs différents. C'est le cas des applications HPC de type fork/join comme le sont les codes OpenMP.

Nous proposons dans ce chapitre de mesurer et optimiser le surcoût en temps d'exécution lié à la virtualisation afin de fournir un mécanisme de négociation de ressources entre des applications multithreadées concurrentes à l'aide de machines virtuelles. L'utilisation de machines virtuelles est motivée par l'utilisation d'environnement personnalisé et la migration d'application s'exécutant au sein de ressources préalablement réservées (job) au sein d'un cluster HPC dont la configuration est peu flexible et uniforme entre les nœuds de calcul. L'objectif est de réaliser un gestionnaire dynamique de ressources au sein d'un nœud de calcul basé sur l'abstraction des ressources induite par la virtualisation. Pour cela, nous étudions notamment le surcoût de code multithreadé exécuté au sein de machines virtuelles. Enfin, pour illustrer un cas d'application concrète d'utilisation de ce gestionnaire, nous présentons un mécanisme de négociation entre applications OpenMP concurrentes et en étudions les performances.

2.1 Gestion dynamique des ressources d'une application HPC

Déterminer en temps réel les ressources optimales à fournir à une application parallèle est une tâche souvent complexe et dépendante de nombreux paramètres. Le jeu d'entrée et l'environnement d'exécution du programme sur une machine exécutant des applications concurrentes entrainent une variabilité des performances et temps d'exécution d'une application (contention réseau par exemple) et donc son degré de parallélisme. Afin de permettre à l'utilisateur d'exploiter au mieux les ressources de calcul (CPU) qu'il alloue à une ou plusieurs applications multithreadées au fil de leur exécution, nous proposons un outil basé sur l'utilisation de machines virtuelles. Notre outil vise à fournir des environnements virtuels optimisés aux développeurs et utilisateurs d'applications scientifiques HPC afin de moduler et maximiser en terme d'utilisation les CPU mis à la disposition de chaque application. Dans cette section, nous présentons les trois aspects permettant la mise en œuvre de notre proposition que sont la minimisation du surcoût lié à la virtualisation, la gestion dynamique des CPU d'un nœud de calcul attribués à une machine virtuelle ainsi que le partitionnement dynamique de ces CPU.

2.1.1 Minimiser le surcoût lié à la virtualisation

Avant de parler de service ou de partitionnement à l'aide de la virtualisation, il est essentiel de connaître précisément le surcoût lié à la virtualisation et plus spécifiquement pour des applications de type calcul haute performance. La virtualisation et le calcul haute performance semblent a priori deux domaines intrinsèquement antagonistes. En effet, la virtualisation consiste en une gestion haut niveau de ressources matérielles afin de moduler le nombre de tâches traitées par un service. Nous parlerons de programme *embarassingly parallel*, c'est-à-dire que les tâches sont souvent séquentielles et sans communication ou dépendance. À l'inverse, le calcul haute performance utilise une connaissance fine de la hiérarchie des ressources physiques dédiées à l'exécution d'un code scientifique. Les codes scientifiques de simulation numérique ont besoin de répartir leur algorithme de résolution et les données sur un nœud de calcul. Afin d'obtenir des performances en temps d'exécution, les données doivent être placées finement de manière à minimiser les temps d'accès mémoire. Le plus souvent, ces tâches sont dépendantes les unes des autres et ont besoin de communications provoquant un surcoût en temps d'exécution de la simulation numérique. Ces communications doivent être optimisées d'une part grâce à des algorithmes de communication efficaces et d'autre part au travers d'un placement adéquat des tâches. Cette antinomie de besoins entre l'informatique en nuage basé sur des techniques de virtualisation et le calcul haute performance était, au début de notre travail de thèse, un frein majeur à l'utilisation de machines virtuelles. Des travaux ont été menés sur les machines virtuelles monocœurs et l'impact de leur utilisation pour des applications de type MPI. Mais les machines virtuelles multicœurs ainsi que leurs performances étaient peu étudiées en contexte de calcul haute performance. De nombreuses questions se sont alors posées. Quel est le surcoût en temps d'exécution lié à l'emploi d'une machine virtuelle multicœur ? Quels sont les besoins du calcul haute performance vis-à-vis de la virtualisation ? Quels sont les paramètres d'un hyperviseur permettant de répondre à ces besoins ? Nous avons donc besoin d'étudier un large panel d'applications scientifiques et de connaître de façon approfondie l'hyperviseur afin de fournir un outil générique de création de machines virtuelles dédiées au domaine du calcul haute performance.

2.1.2 Contrôler les ressources attribuées à un code de calcul

Une fois la configuration optimale d'une machine virtuelle HPC connue, nous avons besoin de comprendre comment ajouter ou diminuer le nombre de CPU d'une application scientifique. Pour cela, nous modulons les ressources exposées par la machine virtuelle et nous nous assurons que le support exécutif qui réalise notre simulation numérique s'y adapte efficacement. Nous rappelons que notre objectif est de fournir des environnements virtuels permettant un fort couplage entre le nombre de CPU exposés à l'application et sa capacité à les utiliser. Cette capacité à utiliser un nombre croissant de CPU rejoint la notion de degré de parallélisme, nous considérons ici le degré de parallélisme local à des sections d'une application. La principale difficulté de ce couplage réside dans l'optimisation d'un mécanisme de redimensionnement de machine virtuelle respectant les contraintes inhérentes au calcul haute performance.

Notre objectif est d'adapter le comportement du support exécutif au sein de la machine virtuelle pour qu'il soit capable de s'adapter à la variation de ressources. En effet, le fonctionnement standard de la plupart des supports exécutifs (Intel OpenMP RTL, GOMP, MPICH, OpenMPI, Intel TBB, ...) consiste à découvrir les ressources mises à sa disposition au moment de son initialisation. Cette découverte peut être automatique ou bien guidée par l'utilisateur. Les ressources ainsi découvertes sont alors considérées comme acquises pour la durée de l'exécution. Il est ensuite de notre initiative de notifier les changements de ressources afin de permettre au support exécutif de moduler son degré de parallélisme. De notre point de vue, il est important que ces notifications soient non intrusives c'est-à-dire qu'elles ne nécessitent pas de modifier le code du support exécutif. Ces notifications pourront alors s'appliquer de manière générique à toutes les implémentations d'un même standard ou d'une même norme de calcul parallèle. Dans le cas d'OpenMP par exemple, les notifications devront servir pour les supports d'exécution Intel OpenMP, GOMP ou encore MPC.

Pour réaliser des machines virtuelles dynamiques, il est donc nécessaire de réfléchir à la manière dont sont partitionnées les ressources entre les machines virtuelles en prenant en compte l'architecture matérielle de l'hôte. Mais il faut également fournir au support exécutif les notifications nécessaires pour qu'il adapte son degré de parallélisme aux ressources à sa disposition.

2.1.3 Partitionner un nœud de calcul à l'aide de la virtualisation

Nous avons considéré l'usage de machines virtuelles et non de conteneurs plus légers comme les cgroups afin de permettre de fournir des environnements spécifiques et définis par un utilisateur. Les conteneurs permettent de limiter le surcoût de l'isolation des ressources mais, comme nous l'avons vu dans le chapitre précédent, ne permettent pas d'exécuter un système complet. Nous avons montré que les machines virtuelles peuvent être vues comme des sous-ensembles de ressources virtualisées dynamiques. Côté invité, un système d'exploitation standard permet de fournir à l'utilisateur un environnement propre au déploiement de son code de calcul. Côté hôte, une machine virtuelle est un processus standard avec de la mémoire et des *threads*. Cette abstraction côté hôte permet de mesurer l'activité globale d'une machine virtuelle : hyperviseur, système d'exploitation ainsi que programmes exécutés. On peut mesurer la consommation globale en mémoire et en nombre d'instructions lors de l'exécution d'un code de simulation. Cette mesure présente donc l'avantage d'être indépendante du modèle de parallélisme choisi et ne demande aucune connaissance du code de calcul. Partant de cette mesure, nous essayons de déduire le degré de parallélisme en temps réel d'un code de simulation. Ce degré de parallélisme, nous permettant par la suite de répartir les ressources de calcul entre

plusieurs applications concurrentes au sein d'un nœud de calcul. Ce service peut être vu comme une alternative à l'allocation statique de ressources telle qu'elle est pratiquée en général dans les grands centres de calcul. Le principe est de favoriser les applications les plus parallèles vis-à-vis des moins parallèles. Ce principe rejoint l'idée d'efficacité, une machine virtuelle n'exploitant pas suffisamment le taux d'instruction réalisable par un CPU se verra attribuer moins de CPU au profit d'une autre réalisant un plus haut taux d'instruction. Cette information n'étant pas triviale et en général dépendante du jeu d'entrée, elle peut être complexe voire impossible à connaître pour l'utilisateur final du programme de simulation. De plus, il est important de pouvoir adapter finement les ressources nécessaires aux différentes phases d'un calcul scientifique. En effet, il n'est pas rare qu'un code de simulation alterne des phases de calcul ou d'écriture de données dont les besoins sont très hétérogènes.

2.2 Machine virtuelle dédiée à des applications de calcul haute performance

Pour estimer le surcoût en temps d'exécution, nous avons réalisé des tests sur un nœud de calcul doté de deux sockets de processeurs Nehalem quadricœurs. Nous avons représenté son architecture sur la figure 2.1. On peut voir que le surcoût le plus important a lieu quand on utilise huit threads. Dans cette situation la machine virtuelle utilise les deux sockets et donc deux nœuds NUMA.

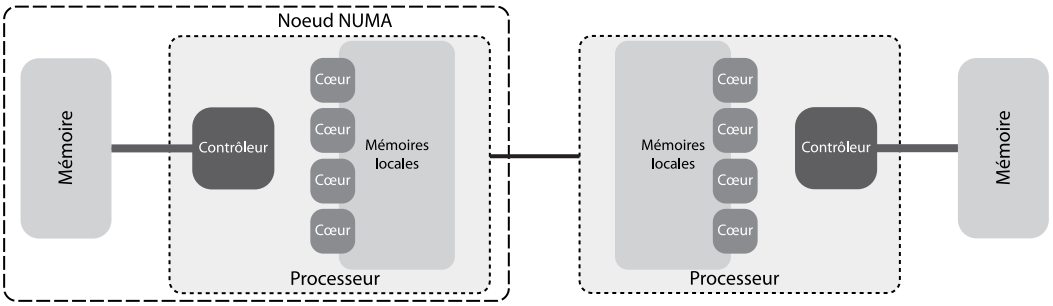


FIGURE 2.1 – Architecture d'un nœud utilisé pour nos tests

Comme nous pouvons le constater sur la figure 2.2, le temps d'exécution d'applications au sein de machines virtuelles peut entrainer un surcoût en temps d'exécution important. Lors de nos premiers résultats sur les benchmarks OpenMP NAS, nous avons utilisé la configuration par défaut de l'hyperviseur QEMU. Il est donc nécessaire d'identifier la cause de ce surcoût et de le minimiser à l'aide des paramètres de configuration à notre disposition. Dans cette section, nous ciblerons tout particulièrement les aspects de localité mémoire et de table des pages dans le cas de l'hyperviseur QEMU.

Une machine virtuelle tout comme une machine physique a besoin de ressources physiques pour fonctionner. Dans le cas d'une machine virtuelle, les ressources (mémoire, CPU) vues par le système invité sont des éléments virtuels créées par l'hyperviseur. Ce fonctionnement permet entre autres à l'utilisateur de créer une architecture matérielle virtuelle sans relation directe avec l'architecture matérielle physique. Nous nous sommes donc intéressés aux liens existants par défaut entre les ressources virtuelles et physiques et plus spécifiquement en terme de mémoire et CPU.

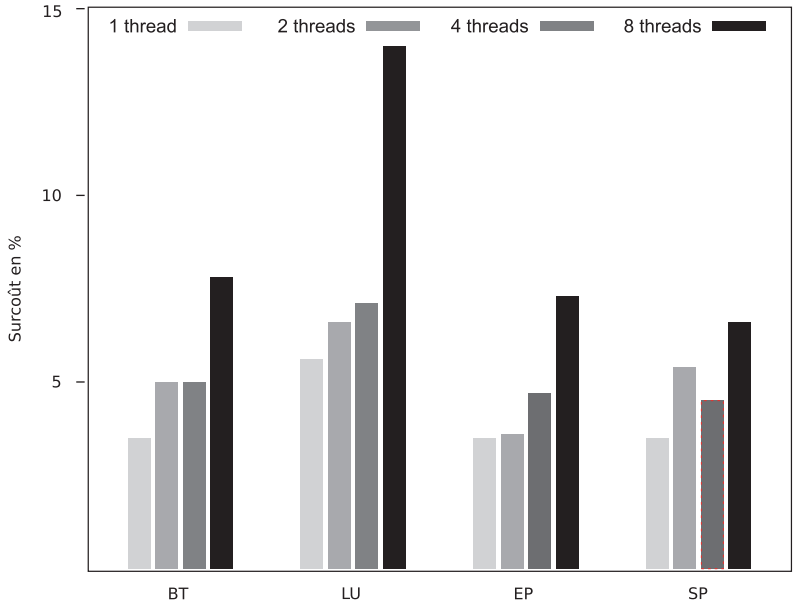


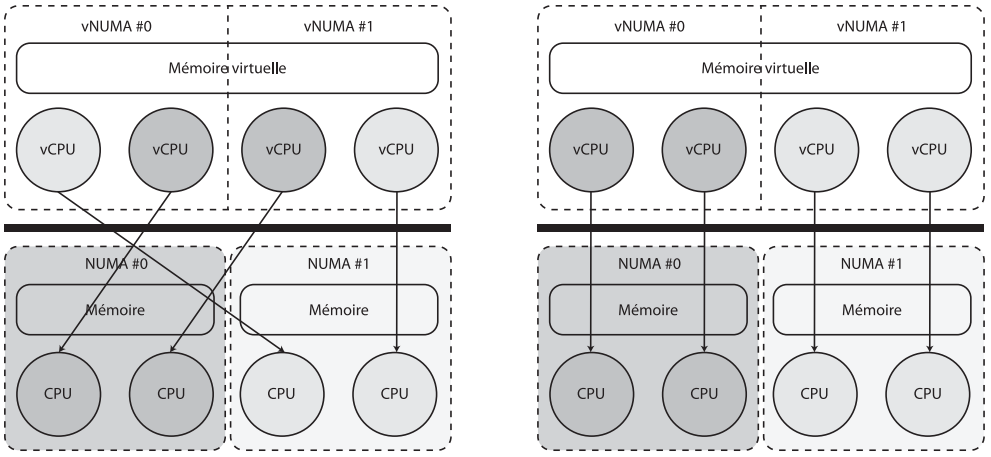
FIGURE 2.2 – Surcoût en temps d'exécution du code NAS OpenMP

2.2.1 Machine virtuelle et CPU virtuel

La connaissance des informations relatives à la topologie matérielle d'une machine est primordiale dans l'utilisation de codes scientifiques. Cette découverte de la hiérarchie d'un nœud de calcul peut être réalisée par le biais d'un outil comme par exemple *HWLOC* qui est utilisé par certains supports exécutifs MPI (OpenMPI, MPC ...). D'autres supports exécutifs, comme celui d'Intel pour sa bibliothèque OpenMP, possèdent leur propre mécanisme de reconnaissance. Pour autant, la principale différence entre ces implémentations réside dans la portabilité sur un grand nombre d'environnements de simulation (Linux, Mac, Windows). Ce sont toujours les fichiers fournis par le système d'exploitation qui permettent de décrire les éléments matériels à sa disposition et leur hiérarchisation. Par conséquent, il suffit d'exposer au système d'exploitation invité la bonne topologie ou sous-topologie pour obtenir un déploiement adéquat d'un support exécutif parallèle. Pour cela, les hyperviseurs permettent à l'utilisateur de préciser la topologie matérielle qu'il souhaite exposer côté invité. Dans la plupart des hyperviseurs, grâce aux paramètres de configuration, il est possible de préciser le nombre de sockets et de nœuds NUMA ainsi que les emplacements mémoire associés et le nombre de cœurs de calcul.

L'utilisateur peut donc construire une topologie personnalisée sans aucune réalité physique. Cette absence de contrainte entraine une impossibilité pour l'hyperviseur d'établir une table de correspondance entre topologie émulée et topologie physique. Le placement des cœurs de calcul virtuels revient donc à la charge de l'utilisateur s'il le souhaite. Dans notre cas, nous réalisons un partitionnement de la topologie matérielle. Nous sommes donc capables de construire de manière déterministe une bijection entre les VCPU et les CPU physiques afin d'obtenir un couplage fort entre réalité physique et virtuelle. Pour QEMU par exemple, on peut utiliser les options *-smp* ou *-Numa*. Nous avons donc réalisé un premier outil basé sur les outils HWLOC et QEMU qui permettent de généraliser la création des paramètres topologiques de la machine virtuelle et de placer les VCPU sur les CPU physiques.

Les résultats obtenus grâce à cet outil sont très proches des résultats précédents et ne permettent pas de diminuer de manière importante le surcoût induit par la virtuali-



(a) Configuration par défaut (b) Configuration manuelle

FIGURE 2.3 – Couplage CPU d'une topologie émulée avec la topologie physique

sation. Nous nous sommes donc intéressés au placement de la mémoire d'une machine virtuelle.

2.2.2 RAM et placement mémoire

De la même manière que pour le placement des entités de calcul, il est à la charge de l'utilisateur de déterminer à quelle(s) mémoire(s) physique(s) est associée la mémoire d'une machine virtuelle. Au moment de nos évaluations, l'hyperviseur QEMU que nous utilisions pour la création de nos machines virtuelles ne possédait pas d'option de placement mémoire pour la RAM de la machine virtuelle. Nous avons donc étudié le mécanisme d'initialisation et allocation de la RAM d'une machine virtuelle par QEMU.

Le segment de mémoire alloué par QEMU pour former la RAM de la machine virtuelle est réservé par deux mécanismes en fonction du type de pages utilisées (petite page ou grosse page). Dans le cas des petites pages, l'allocation est réalisée à l'aide de la *glib*. L'espace réservé est alors virtuel et devient physique au gré des accès mémoire réalisés (i.e. *first-touch*) par les threads de QEMU. Une partie des pages est ainsi touchée par le thread qui initialise QEMU à son lancement. Le mécanisme de *first-touch* combiné à l'absence de *binding* de ce thread entraine une répartition aléatoire d'une partie des pages physiques utilisées par la machine virtuelle. Pour les grosses pages, l'allocation se fait par le biais d'un appel système *mmap* suivi d'un appel système *truncate* pour en assurer la bonne taille. L'utilisation de *truncate* déclenche une écriture sur toutes les pages allouées par *mmap*. QEMU réalise un appel *truncate* car l'utilisation des grosses pages se fait par le biais d'un fichier. De la même manière, on se retrouve avec une RAM allouée en priorité sur le nœud NUMA du thread d'initialisation de QEMU. Ce comportement aléatoire entrainait pour le cas des petites pages un écart important entre le temps d'exécution de la première répétition de nos benchmarks et les suivantes. En effet, la première exécution s'effectuait sur une RAM quasiment vierge c'est-à-dire dont l'allocation était purement virtuelle. Les défauts de page étaient donc réalisés par le VCPU qui réalisait l'allocation, le mécanisme de *first-touch* était donc pleinement efficace. Les exécutions suivantes se voyant attribuer des pages déjà présentes dans la RAM de la machine virtuelle, nous perdions donc le placement mémoire des pages utilisées par l'application.

Nous avons donc étudié la manière dont QEMU expose la hiérarchie mémoire émulée au système invité. La première observation est la nature contiguë de l'espace mémoire alloué pour former la RAM de la machine virtuelle et ce dans les deux cas de figure. Cette mémoire est partitionnée en un nombre de blocs de même taille. On a autant de blocs que de nœuds NUMA précisés par l'utilisateur. Nous avons donc ajouté une affinité mémoire différente pour chacun des blocs mémoires. Cela permet aux VCPU de profiter d'une bonne localité mémoire et de limiter leurs accès à la mémoire d'un nœud distant.

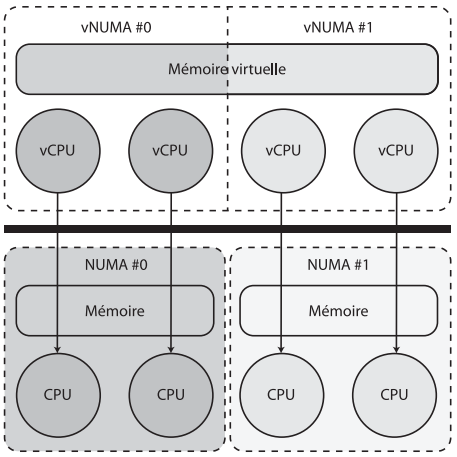


FIGURE 2.4 – Couplage complet d'une topologie émulée avec une topologie physique

Cette modification de QEMU a été réalisée avec la bibliothèque LibNUMA. Cette bibliothèque permet de gérer la politique d'allocation mémoire. Elle fournit un ensemble de primitives permettant de migrer ou d'imposer le futur placement des pages allouées par un processus. Le choix de la libNUMA est motivé par le fait que l'on veut uniquement modifier la politique d'allocation des pages. Notre approche était purement expérimentale et la bibliothèque libNUMA possède très peu de contraintes aussi bien pour son installation que pour son ajout à la compilation de QEMU. Parmi les fonctionnalités de libNUMA, nous avons utilisé la primitive *mbind* qui permet de modifier la politique de placement d'une page système. L'attribution d'une page physique lors d'un défaut de page peut être alors différente de la politique first-touch. On réalise un appel à *mbind* par nœud NUMA émulé par QEMU. Lors du *binding* des pages allouées par QEMU, on spécifie la politique que l'on souhaite appliquer à l'ensemble des pages d'un nœud VNUMA . Nous avons choisi le mode `MPOL_BIND`. Ce mode impose que les pages soient placées sur le ou les nœuds définis par l'utilisateur. Cette politique n'a pas d'effet sur les pages ayant déjà été touchées. Dans ce cas, on peut préciser à *mbind* le comportement à adopter à l'aide d'une option : `MPOL_MF_MOV`. Cette option demande au noyau de migrer toutes les pages déjà existantes d'un processus vers les nœuds NUMA qui satisfont la politique déclarée dans le mode. À noter que les pages partagées par plusieurs processus ne seront pas migrées avec ce flag, mais ce n'est pas notre cas ici. Dans le cas des grosses pages, il est nécessaire de demander la migration des pages. En effet dans ce mode d'allocation les pages sont directement touchées par l'appel à *truncate*.

La figure 2.5 montre le surcoût en temps d'exécution du NAS OpenMP BT de classe B. Comme on peut le constater, le surcoût observé en introduction de cette section a diminué. Ce résultat valide notre approche et nos hypothèses concernant la cause du surcoût important observé lors de nos premiers tests.

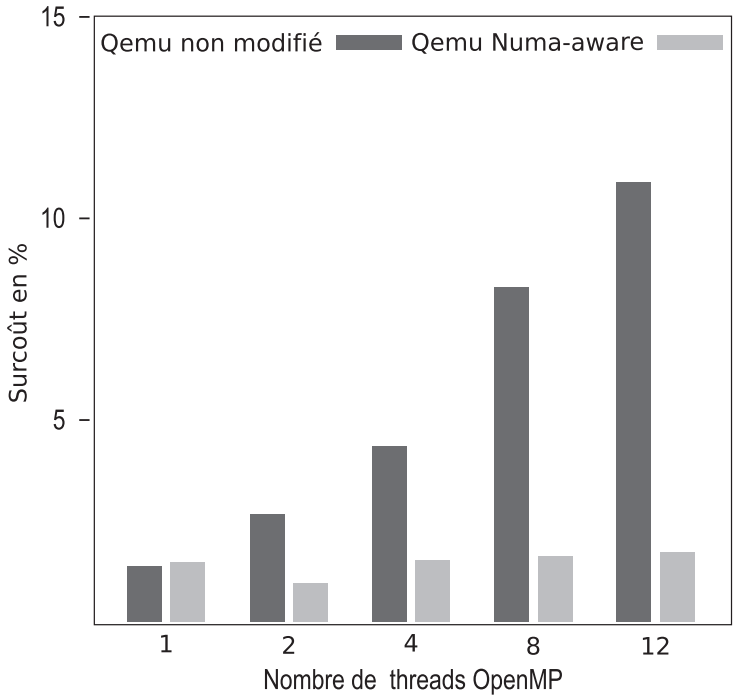


FIGURE 2.5 – Surcoût en temps d'exécution du code NAS OpenMP BT

```
-object memory-backend-ram, size=512M, policy=bind, host-nodes=1,
id=ram-node1 -numa node, nodeid=1, cpus=8-15, memdev=ram-node1
```

FIGURE 2.6 – Allocation d'une RAM de 512 Mo attachée au nœud NUMA 1 de l'hôte

À la différence du placement des VCPU, le placement mémoire ne peut pas être réalisé en dehors de QEMU par le biais d'outil générique. Nous avons montré qu'il est important de pouvoir configurer ce placement et souligné l'absence de paramètre pour le faire.

2.2.3 Nouveauté dans QEMU et configuration HPC

Même si nous ne sommes pas à l'origine du patch réalisé dans QEMU, l'évolution de l'hyperviseur QEMU vient valider les mesures et réflexions que nous avons énoncées précédemment. En effet, la version 2.10 de QEMU sortie le 1er août 2014 intègre un outil de punaisage de la mémoire pour les architectures x86 de type NUMA. L'utilisateur peut dorénavant créer une RAM en plusieurs segments, les `memory-backend-ram` dont on peut spécifier pour la politique de placement. L'utilisation conjointe de l'option Numa et d'un objet `memory-backend-ram` permet de retrouver un fort couplage entre une topologie invitée et une topologie hôte. Le placement des VCPU reste quant à lui à la charge de l'utilisateur.

2.3 Gestion dynamique des CPU d'une application scientifique

Dans cette section, nous allons décrire le mécanisme mis en place pour faire varier les ressources d'une machine virtuelle. L'objectif est de retirer et ajouter dynamiquement un ou plusieurs cœurs de calcul d'une machine virtuelle.

Nous avons vu dans le chapitre 1 que les cœurs physiques d'une machine virtuelle utilisant l'hyperviseur KVM sont des threads créés et ordonnancés par l'hyperviseur. Ces threads s'exécutent sur des cœurs physiques du nœud de calcul. Les VCPU s'exécutent par le biais du module noyau KVM dans un mode invité afin de réduire au maximum le coût de la virtualisation.

La gestion dynamique d'un cœur de calcul invité nous amène à considérer le fonctionnement de l'hyperviseur qui est en charge d'exposer les ressources virtuelles au noyau invité. Pour cela, nous nous sommes intéressés au fonctionnement du « hotplug » qui consiste à ajouter un périphérique émulé à une machine virtuelle. Ce mécanisme se fait à chaud c'est-à-dire sans relancer l'hyperviseur ou le système invité. De la même manière, on parle de « hotunplug » pour l'opération qui retire un périphérique émulé à une machine virtuelle sans la redémarrer. Bien que ces mécanismes existent de manière générique dans de nombreux hyperviseurs, nous les avons considérés plus particulièrement sur QEMU. Les CPU physiques sont détectés et exposés au système d'exploitation par le biais du BIOS. Le BIOS permet de personnaliser l'initialisation des périphériques détectés tels que le nombre de cœurs maximum d'un processeur ou encore l'activation des extensions matérielles pour la virtualisation. Par conséquent, un noyau ne peut pas utiliser un cœur non découvert préalablement par le BIOS. Il est donc nécessaire de préciser à l'hyperviseur le nombre de cœurs maximum qui sera utilisé par une machine virtuelle dès son lancement. Ainsi, le noyau est initialisé avec le nombre maximum de CPU définis. Les CPU en trop sont démarrés avec le statut déconnecté, c'est-à-dire inutilisables pour le système d'exploitation et pour les autres étant connectés. Il est ensuite possible pour un utilisateur privilégié de la machine virtuelle de changer le statut d'un CPU de offline vers online ou l'inverse. Il est toutefois impossible d'éteindre le CPU numéro 0, afin de se prémunir d'un état incohérent que serait un système sans aucune ressource de calcul. Une action complète de *hotplug* a donc besoin d'agir à la fois au niveau de l'hyperviseur pour initialiser et démarrer un VCPU et au niveau du noyau invité afin de notifier l'ajout du CPU au noyau invité.

2.3.1 Interaction à chaud avec l'hyperviseur

L'hyperviseur QEMU fournit une API permettant d'interagir directement avec l'hyperviseur sans redémarrer la machine virtuelle. Cette API permet de réaliser des requêtes *JavaScript Object Notation (json)* à destination de l'hyperviseur. Ces requêtes peuvent être de simples demandes telles que le type de CPU virtualisé ou l'identifiant de threads des CPU de l'invité ou l'ajout à chaud de matériel tel qu'une carte réseau. Parmi les fonctions de l'API, on trouve aussi l'ajout et la suppression de CPU de la machine virtuelle. Lors de nos premiers travaux, le support *hotplug* n'était pas complet. Il était possible de réaliser une action de *hotplug* mais pas de *hotunplug*. On ne pouvait alors pas entièrement gérer les cœurs d'un invité depuis l'hôte au travers de QEMU.

Nous avons donc décidé de ne pas utiliser QMP, mais de mettre en place une synchronisation entre noyau hôte et invité. Le support étant aujourd'hui complet, il serait intéressant de voir dans quelle mesure l'utilisation de cette API pourrait être intégrée dans notre gestionnaire. De plus, il faudrait mesurer le coût et la latence de traitement d'une requête vis-à-vis des mécanismes que nous avons mis en place et que nous détaillons dans la suite du chapitre.

2.3.2 Gestion dynamique des CPU virtuels

Pour moduler les ressources utilisées par le système invité, nous avons besoin de faire varier son nombre de CPU. Comme nous l'avons vu, il est possible d'instancier un

nombre arbitraire de CPU au sein de l'invité. Nous pouvons ensuite moduler le nombre de CPU vu par les programmes s'exécutant sur l'invité en modulant le statut connecté (actif) ou déconnecté (inactif) d'un CPU. Cette modulation est possible par le biais de fonctions fournies par le noyau UNIX de l'invité. Cette solution a l'avantage de ne pas nécessiter d'action de la part de l'hyperviseur. Par contre, les VCPU de tous les CPU instanciés au démarrage de l'invité peuvent être potentiellement ordonnancés malgré leur statut déconnecté. Nous avons donc mesuré le surcoût en temps d'exécution des VCPU excédentaires.

Comme on peut le voir sur la courbe, les VCPU des CPU déconnectés n'ont pas d'influence sur le temps d'exécution de notre benchmark OpenMP. Ce résultat valide la pertinence de notre approche qui consiste à éviter de faire appel à l'hyperviseur afin de réaliser des synchronisations directement entre processus UNIX sur le système hôte et le noyau du système invité. Nous avons donc basé notre architecture sur plusieurs acteurs que sont le gestionnaire de ressources et les agents topologiques. Le gestionnaire de ressources est un processus UNIX présent au sein de l'espace utilisateur. Son rôle est d'attribuer une liste de ressources à une machine virtuelle. Pour cela, il envoie des requêtes aux agents topologiques présents dans les machines virtuelles qu'il supervise.

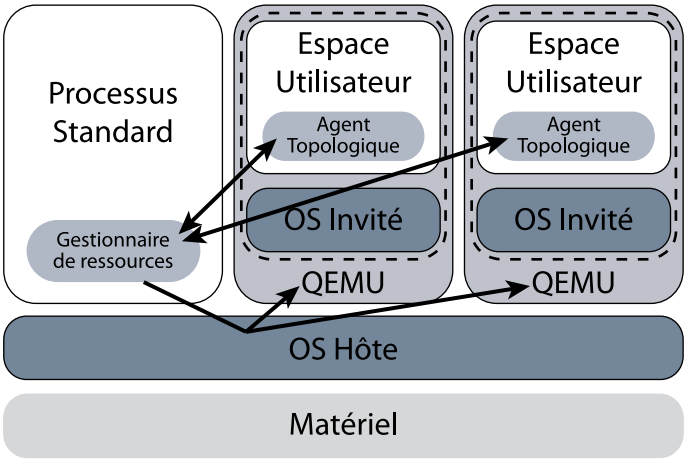


FIGURE 2.7 – Architecture globale de notre outil

Nous avons mis en place les outils permettant de gérer dynamiquement les ressources d'une machine virtuelle. Maintenant nous allons nous intéresser à la manière dont le gestionnaire de ressources répartit les ressources entre les machines virtuelles.

2.3.3 Distribution fair-play des CPU

Nous avons vu que le critère du placement mémoire sur des machines NUMA est essentiel pour obtenir de bonnes performances pour un code multithreadé. Il est évident que modifier les ressources attribuées à une machine virtuelle induit une perte de localité mémoire. En effet, lorsque le nombre de cœurs diminue alors les cœurs peuvent être amenés à réaliser des accès au nœud NUMA distant. De plus, si le gestionnaire de ressources attribue des listes disjointes de CPU entre deux partitionnements alors la localité mémoire sera mauvaise. Nous nous sommes donc intéressés tout particulièrement au mécanisme de création des listes de CPU pour une machine virtuelle. Nos critères sont les suivants :

1. Minimiser le nombre de machines virtuelles sur un même nœud NUMA
2. Maximiser le nombre de CPU commun à deux attributions successives

Pour répondre à ce besoin, nous avons mis en place une notion d'affinité entre CPU et machine virtuelle. Cela consiste à construire la liste des CPU gérés par le gestionnaire de ressources par ordre décroissant de priorité pour chaque machine virtuelle. Nous présentons dans la figure le principe de la création de ces listes pour le cas de quatre machines virtuelles réparties sur deux nœuds NUMA de deux cœurs.

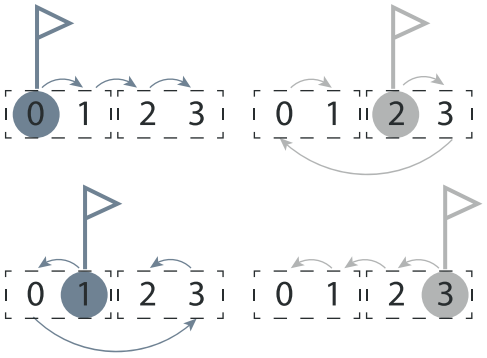


FIGURE 2.8 – Construction des listes d'affinité CPU

Notre problème est le suivant, nous avons des nœuds NUMA et un nombre de cœurs par nœud NUMA et r le ratio de machines virtuelles actives simultanément sur un nœud de calcul. Il est évident pour que assurer une persistance dans la liste des CPU attribués à une machine virtuelle, nous avons besoin d'un algorithme déterministe. Pour obtenir une répartition des CPU au sein des différentes machines virtuelles les plus stables possible, il faut limiter la compétition lors de l'attribution. Nous avons donc créé des listes de CPU tel que le rang de la première collision entre deux listes CPU soit en moyenne le plus grand possible.

L'avantage de ce placement est qu'il est facile de calculer la liste de chaque machine virtuelle si on connait le nombre de cœurs assignés à chaque machine virtuelle. Ce calcul peut être fait de manière indépendante pour chaque machine virtuelle. De plus, cette méthode nous permet de garantir que la variation des cœurs alloués entre deux partitionnements soit relativement faible.

2.4 Mesure d'efficacité d'un code OpenMP

Maintenant que nous avons vu comment contrôler l'accès aux CPU par une machine virtuelle, nous allons détailler la partie efficacité de notre algorithme. L'objectif consiste à surveiller l'activité des VCPU afin de garantir qu'ils utilisent efficacement les CPU physiques mis à la disposition d'une machine virtuelle. Nous détaillons dans cette section la mise en place de notre outil de monitoring des VCPU d'une machine virtuelle. Afin de montrer la pertinence de notre approche, les résultats de cette section seront ceux de benchmarks exécutés sans machine virtuelle. Le surcoût pour le monitoring VCPU étant forcément plus important que pour des threads utilisateurs, le surcoût en natif doit être négligeable pour être étendu aux VCPU d'une machine virtuelle.

2.4.1 Monitoring non intrusif

Afin de ne pas être intrusifs, nous avons besoin d'une interface générique qui permet d'injecter des fonctions ou actions au déroulement classique du programme utilisateur. Nous avons pour cela considéré deux approches que sont l'utilisation d'une API de débug

spécifique à OpenMP : OMPT et l'interception de symboles pour des bibliothèques dynamiques.

2.4.1.1 Interception de bibliothèques dynamiques

Pour insérer du code dans un programme déjà compilé, nous nous sommes basés sur le fonctionnement des bibliothèques dynamiques afin d'intercepter des appels de fonctions particuliers. Pour cela, il nous faut comprendre comment sont utilisées une bibliothèque dynamique par un programme utilisateur ainsi que les fonctions permettant d'en modifier le comportement.

Editeur de lien dynamique (dynamic linker) Lors de la compilation d'un programme, l'édition de liens est une étape permettant de créer des fichiers exécutables ou des bibliothèques dynamiques ou statiques, à partir de fichiers objet. Les fichiers objet sont des fichiers intermédiaires créés lors de l'étape précédant l'édition de liens et qui contiennent du code machine. Les fonctions sont des symboles qui sont soit internes aux fichiers soit externes. Lorsque le symbole est externe, l'éditeur de lien recherche le fichier objet ou la bibliothèque à laquelle il se rapporte puis crée un lien entre le symbole externe et le symbole trouvé dans les autres fichiers. L'édition de lien est soit statique soit dynamique. Dans le cas statique, le fichier objet et la bibliothèque sont liés dans le même fichier alors que dans le cas dynamique la bibliothèque n'est pas contenue dans le fichier exécutable. Les liens entre les symboles d'une application compilée dynamiquement ne sont pas fixés lors de la compilation mais établis lors du lancement de l'exécutable. Les commandes Linux `ldd` et `nm` , nous permettent de connaître respectivement les bibliothèques nécessaires au programme et les symboles qu'elles contiennent. La compilation est très rarement statique pour des raisons de taille de programme. Par exemple, un simple programme tel que le classique helloworld (ci-dessous) possède une taille de 8 ko quand il est compilé dynamiquement et 828 ko dans sa version statique. On peut donc considérer qu'il n'est pas utile dans un premier temps de se soucier des programmes compilés statiquement. Nous nous intéresserons donc uniquement aux programmes dont la bibliothèque du support exécutif parallèle n'est pas liée statiquement.

Helloworld

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
5 {
6       printf("Hello world\n");
7       return EXIT_SUCCESS;
8 }
```

Chargement et pré-chargement de bibliothèque dynamique L'utilisation de bibliothèques dynamiques entraine un chargement des bibliothèques au démarrage du programme. Pour cela, le programme va chercher dans les bibliothèques dont les chemins sont répertoriés dans la variable d'environnement `LD_LIBRARY_PATH`. Dès que l'ensemble des bibliothèques nécessaires au programme sont trouvées, les symboles sont liés et le programme démarre. Ce mécanisme nous permet de nous placer de manière transparente entre le programme exécutable de l'utilisateur et l'exécution de la bibliothèque de calcul parallèle qu'il a choisie. Pour cela, on utilise la variable d'environnement `LD_PRELOAD` qui nous permet de spécifier une bibliothèque partagée qui sera chargée au début du programme avant toutes autres bibliothèques. On peut ainsi intercepter un

lien entre le symbole classique et sa bibliothèque originelle pour y placer notre propre fonction.

Redirection du symbole intercepté En interceptant le symbole, nous avons cassé le lien vers la bibliothèque qui fournit la vraie fonction. Il faut donc le retrouver pour garantir une exécution adéquate du programme utilisateur. Pour cela, on peut utiliser les fonctions de l'éditeur de lien dynamique pour ouvrir une bibliothèque dynamique et y trouver la fonction voulue. La vraie fonction est ensuite appelée et le flot d'exécution du programme conservé.

Pthread_exemple.c

```
1 int main(int argc,char *argv[])
2 {
3     pthread_t tid;
4     pthread_create(&tid, NULL, fn, NULL);
5     pthread_join(tid, NULL);
6     return EXIT_SUCCESS;
7 }
```

Pthread_join_intercept.c

```
1  int pthread_join(pthread_t thread, void **retval)
2  {
3      int (*join)(pthread_t,void **);
4      void *handle=dlopen("/lib64/libpthread.so.0",RTLD_LOCAL|RTLD_LAZY);
5      int ret;
6      join=dlsym(handle,"pthread_join");
7      printf("avant pthread_join\n");
8      ret=(*join)(thread,retval);
9      printf("apres pthread_join\n");
10     return ret;
11 }
```

```
$LD_PRELOAD=./pthread_join_intercept.so ./pthread_join_example.exe
avant pthread_join
après pthread_join
```

Dans un premier temps, les appels de fonction qui nous intéressent sont ceux correspondant à la création des threads participant à l'exécution du programme. Comme on peut le voir dans les exemples de code associés à la création d'un thread POSIX, l'utilisation de PRELOAD et *dlfcn* correspond bien à nos besoins et nous permet de surveiller de manière transparente une application à l'aide de fonctions ciblées. Pour autant, cette démarche présente de fortes limitations. Nous devons connaitre le prototype des fonctions à intercepter et leur nom ce qui n'est pas toujours possible dans le cas de bibliothèques propriétaires. En effet, l'API d'un standard est commune à tous les supports exécutifs, par contre aucune règle n'existe concernant l'implémentation interne du support exécutif. La bibliothèque dynamique doit donc être adaptée pour chaque support exécutif et à l'évolution de leur structure interne. Pour pallier ce problème, des standards d'API de debugage et profilling ont été écrits. C'est le cas de PMPI (MPI Standard Profiling Interface) pour les supports exécutifs MPI ou OMPT /OMPD pour les runtimes OpenMP.

2.4.1.2 OpenMP & OMPT

Dans le cas d'OpenMP, une API standard appelée OMPT[EMCS+13] a été proposée et normée. Son implémentation repose sur l'ajout de pointeur de fonctions au sein d'une implémentation OpenMP, l'utilisateur ou l'outil d'analyse de performance peut ensuite écrire les fonctions à charger pour chacun de ces pointeurs de fonctions. Cette démarche avait pour intérêt d'être non-intrusive et portable sur les différentes bibliothèques OpenMP. Lors de nos premiers tests, nous avons observé que les fonctions OMPT étaient uniquement implémentées dans le runtime Intel OpenMP. Nous n'avons donc pas utilisé directement le support OMPT dans le cas de GOMP et du support d'exécution Intel OpenMP au profit d'une interception classique de symbole à l'aide d'une bibliothèque dynamique préchargée.

> The OpenMP runtime invokes this callback in the context of an initial thread just after it initializes the OpenMP runtime for itself, or in the context of a new thread created by the OpenMP runtime system just after the thread initializes itself.

FIGURE 2.9 – Extrait norme OMPT - ompt_event_thread_begin

OMPT est aujourd'hui un standard largement implémenté dans la plupart des supports exécutifs OpenMP tels que ceux des compilateurs Intel, IBM, et OpenUH (compilateur openACC). Pour étendre nos résultats obtenus avec GOMP et le runtime Intel OpenMP, nous avons ajouté les événements de démarrage d'un thread, et de démarrage et de fin de région parallèle au runtime MPC OpenMP. MPC utilisant des threads utilisateurs à la différence de GOMP et Intel, nous avions besoin d'un point d'accroche générique pour conserver une approche transparente.

Maintenant que nous avons vu et mis en place deux mécanismes d'interception d'événement au sein d'un support exécutif, nous allons justifier notre choix de profiling d'une application utilisateur.

2.4.2 Choix de la métrique

Nous rappelons que notre objectif est de quantifier l'efficacité d'un thread, et donc plus globalement d'un programme, à utiliser les CPU qui lui sont assignés. Quantifier le parallélisme d'une application est une tâche complexe, il faut en général observer finement le comportement d'une application pour trouver et diminuer le temps passé dans les points chauds (ou goulots d'étranglements). Nous avons donc déterminé un cadre d'application de notre gestionnaire de ressources CPU. Pour nous permettre une approximation rapide et répétée du degré de parallélisme d'un code de simulation, nous allons uniquement étudier les codes de simulation réalisant essentiellement des opérations flottantes et nous négligerons le problème du placement mémoire. De plus, il est nécessaire que notre évaluation se fasse sans connaissance du code utilisateur et de son mode de parallélisme.

À partir de ces contraintes, nous avons choisi de considérer les FLOPS réalisés par chacun des threads d'un support exécutif à mémoire partagée. Dans nos exemples, nous avons considéré OpenMP, mais ces résultats pourraient être étendus à d'autres standards à la condition qu'ils permettent de moduler le nombre et le placement des threads qu'ils utilisent. C'est le cas par exemple de Cilk ou TBB qui permettent de changer le nombre de *workers* avec la contrainte d'arrêter et de relancer le runtime Cilk. Cette contrainte

est tout à fait similaire à celle d'OpenMP qui doit attendre la fin d'une région parallèle pour modifier son nombre de threads actifs.

2.4.3 PAPI

Nous avons donc besoin d'une bibliothèque permettant de récolter ces informations. Notre choix pour cette bibliothèque était motivé par un besoin de portabilité, la bibliothèque doit donc être disponible pour de nombreuses architectures, mais aussi par l'absence de configuration demandant des droits administrateurs. Enfin, nous voulions une implémentation légère et n'ayant pas un gros impact sur les performances d'un code de calcul. Pour ces raisons, notre choix s'est porté sur la bibliothèque PAPI qui fournit deux niveaux d'implémentation possibles (Low_level et High_level) permettant une configuration plus ou moins fine. Pour cela, l'API PAPI permet d'associer à des threads un ou plusieurs types de compteurs matériels à relever (Défaut de cache L1, Nombre d'opérations à virgule ...), ces compteurs sont ajoutés par l'utilisateur selon ses besoins.

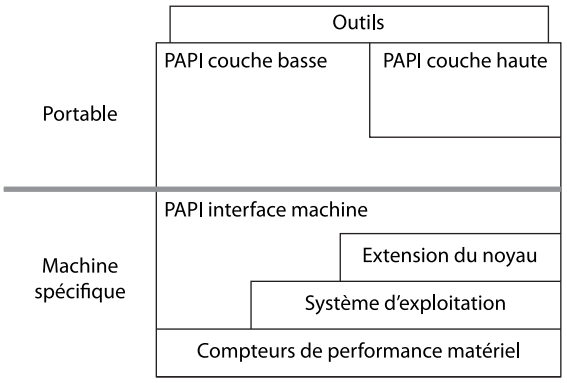


FIGURE 2.10 – Architecture de la bibliothèque PAPI

Notre choix s'est porté sur l'API low_level de la bibliothèque PAPI afin d'avoir le contrôle le plus fin sur l'utilisation des événements PAPI et de leur utilisation. Pour réaliser nos mesures, nous créons un évènement PAPI par VCPU que l'on attache au thread VCPU. Les mesures de performance sont ensuite lues par un thread dédié s'exécutant en surcharge vis-à-vis des autres VCPU. Nous montrons dans la suite dans quelle mesure ce mode de fonctionnement n'entrave pas les performances de nos machines virtuelles. Pour cela, nous allons justifier du choix de la fréquence avec laquelle nous réalisons nos mesures.

2.4.4 Fréquence de rafraichissement

La fréquence de mise à jour de l'activité des threads est un élément primordial de notre algorithme. Nous avons donc étudié son surcoût en temps d'exécution selon deux critères : le surcoût induit par le calcul de l'efficacité calculée à chaque mise à jour et la pertinence des mesures réalisées.

2.4.4.1 Surcoût en temps d'exécution et fréquence de rafraichissement

Notre approche se base sur un monitoring transparent de l'application, nous n'avons donc pas de cœurs dédiés à son activité. Ses calculs se font donc en surcharge avec les autres threads de calcul du code utilisateur. La première idée est donc de chercher la fréquence qui présente un surcoût nul ou acceptable afin de garantir des performances

identiques avec et sans notre monitoring. La figure montre le surcoût en temps d'exécution pour un ensemble de NAS OpenMP exécutés avec 8 cœurs avec le support exécutif GOMP. Comme on peut le constater, on obtient un surcoût de l'ordre de quelques pourcentages avec une fréquence de rafraichissement de 1 ms^{-1}. Nous retrouvons des résultats similaires dans le cas d'une application réelle telle que LULESH (figure 2.12).

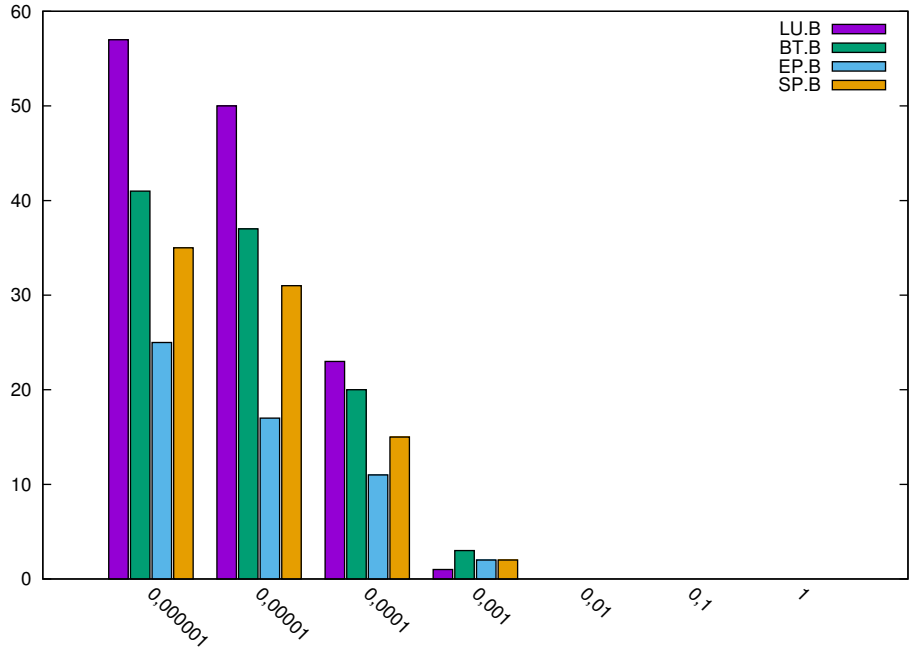


FIGURE 2.11 – Surcoût en temps du monitoring en fonction de la fréquence de polling pour les application NAS OpenMP

2.4.4.2 Fréquence de rafraichissement et pertinence de la mesure

La fréquence de rafraichissement de l'activité des threads est un critère essentiel pour déterminer la réactivité avec laquelle nous pourrons repartitionner les CPU entre les différents codes OpenMP. L'intervalle de polling est réalisé en utilisant la fonction `usleep`, on est donc assuré que l'exécution de notre thread est suspendue pendant au moins 1 *usec*. Il peut être plus long selon l'activité système et la granularité des horloges système.

2.5 Détails d'implémentation

Nous présentons dans cette section les choix d'implémentation que nous avons faits lors du développement de notre outil de gestion de négociation de ressources pour applications OpenMP concurrentes. Notre principale contrainte était de rester le plus générique possible afin de pouvoir tester notre gestionnaire avec plusieurs supports exécutifs OpenMP.

2.5.1 Gestionnaire de ressources et agent topologique

Pour réaliser des machines virtuelles dynamiques, nous avons mis deux acteurs qui permettent de contrôler l'utilisation des ressources physiques attribuées à une machine

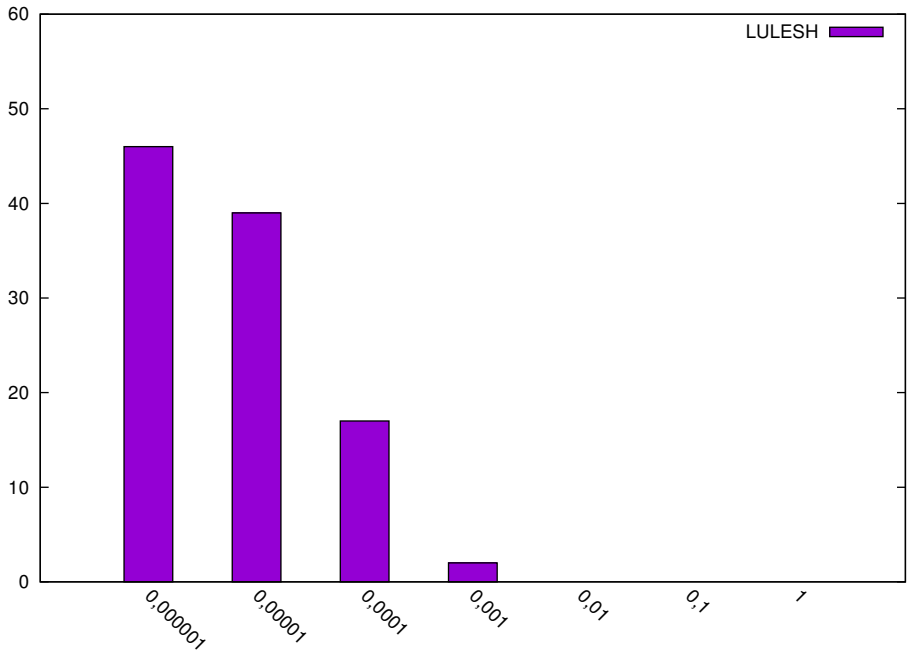
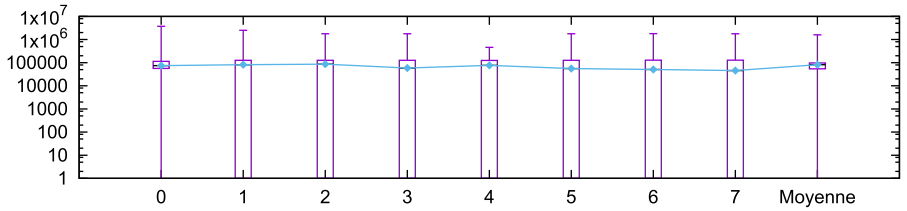
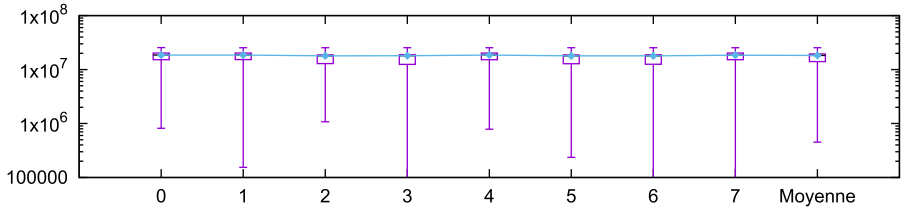


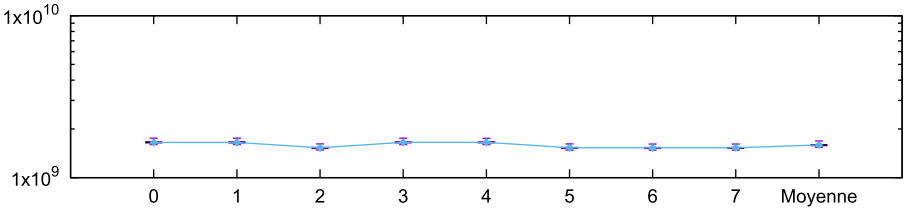
FIGURE 2.12 – Surcoût en temps du monitoring en fonction de la fréquence de polling pour l'application LULESH



(a) Fréquence de polling de 1 usec



(b) Fréquence de polling de 10000 usec



(c) Fréquence de polling de 1 sec

FIGURE 2.13 – Variance des FLOPS mesurés pour chaque thread OpenMP en fonction de la fréquence de polling

virtuelle. D'une part un gestionnaire de ressources qui administre les machines virtuelles et leurs ressources côté hôtes. Il s'agit d'un processus UNIX indépendant des processus QEMU des machines virtuelles qu'il administre. D'autre part un agent topologique dont le rôle est de réaliser les actions demandées par le gestionnaire de ressources. Cet agent s'exécute sous la forme d'un démon lancé dès le démarrage de la machine virtuelle, il communique avec le gestionnaire de ressources sous la forme de requêtes json.

Gestionnaire de ressources Le premier objectif du gestionnaire de ressources est de contrôler le déploiement d'une machine virtuelle. Pour cela, il a besoin de connaître la hiérarchie des ressources mises à sa disposition. Nous utilisons pour cela la bibliothèque *HWLOC* qui permet de découvrir les ressources mises à sa disposition. De plus, cette bibliothèque est souvent présente dans l'ensemble des bibliothèques fournies aux utilisateurs de cluster HPC. Grâce à ces informations, le gestionnaire de ressources génère la liste d'affinité CPU et nœud NUMA en fonction du rang de la machine virtuelle (cf section précédente). Enfin, il ajoute les options QEMU permettant de réaliser la topologie invitée à la ligne de commande QEMU lors de l'ajout d'une nouvelle machine virtuelle au cluster virtuel. De plus, il ajoute le démarrage du service QMP de QEMU par le biais d'un socket UNIX afin de pouvoir échanger des informations avec l'hyperviseur. Le moniteur QMP, nous permet comme nous l'avons expliqué de distinguer les threads QEMU VCPU des threads I/O. Le gestionnaire récupère ainsi la liste des identifiants de threads VCPU pour les assigner aux CPU physiques suivant notre algorithme d'utilisation fair-play des CPU. Nous avons illustré sur la figure 2.14 les choix du gestionnaire de ressources pour une machine virtuelle en fonction de son efficacité.

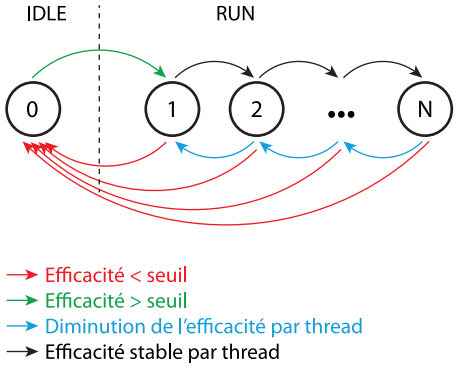


FIGURE 2.14 – Illustration des états d'une machine virtuelle en fonction de son activité

Agent topologique Les actions de l'agent topologique se limitent à deux fonctionnalités qui consistent à mettre à jour ou envoyer le statut des CPU de l'invité. Pour cela, nous avons mis en place un canal de communication entre l'invité et l'hôte utilisant un port série virtio afin de limiter au maximum la latence de ce lien. Avec le port série virto, notre lien de communication reste agnostique du réseau sous-jacent. Côté hôte, nous avons un socket dont le serveur est géré par QEMU et côté invité un périphérique virtio. Les requêtes traitées sont écrites en JSON afin de faciliter l'intégration de nouvelles requêtes, de plus JSON nous a paru un bon choix dans le cadre d'échange de messages entre deux acteurs écrits dans des langages différents. De nombreuses implémentations de *parser* sont disponibles dans de nombreux langages tels que le C, C++ ou Python et garantissent une lecture et une écriture robuste pour nos requêtes.

Lors du traitement d'une requête de mise à jour des CPU de l'invité, l'agent topologique reçoit la liste des CPU dont le statut doit être connecté. Il réalise un filtre

pour connaitre la liste des CPU, dont le statut a changé depuis la précédente requête. Il va ensuite modifier le statut de ce sous-ensemble de CPU en utilisant le fichier online contenu dans le dossier `/sys/devices/system/cpu/cpuX/`. Cette opération demande un privilège administrateur, l'agent topologique est donc en mode privilégié. Pour des raisons de sécurité, il est important d'empêcher l'utilisateur d'une machine virtuelle de pouvoir communiquer avec le démon ou le gestionnaire de ressources. Ainsi, nous n'avons pas mis en place d'API permettant à une application utilisateur de communiquer avec l'agent topologique.

2.5.2 Notifications au support exécutif OpenMP

Nous avons détaillé les mécanismes permettant d'exposer à l'espace utilisateur une variation de ressources CPU au cours du temps. Cette variation sans notification au support exécutif entraine, dans le cas d'une diminution des ressources de calcul, une situation de surcharge. En effet, on a moins de cœurs disponibles pour un nombre de threads inchangé. À l'inverse, on a une perte d'efficacité dans le cas de l'augmentation des ressources, car le support exécutif ne les prendra pas en compte pendant son exécution. Nous avons donc mis en place un mécanisme de variation de ressources conforme à la norme OpenMP, c'est-à-dire que le nombre de threads ne peut varier qu'entre deux régions parallèles consécutives comme l'indique l'extrait de la figure 2.16. Cette situation est illustrée dans la figure 2.15. Les communications entre le gestionnaire et les agents topologiques sont explicites et basées comme nous l'avons expliqué sur des requêtes. Les communications entre agents topologiques et support exécutif OpenMP sont, elles, de nature implicite : l'agent topologique change le nombre de CPU actifs et le support exécutif OpenMP lit le statut des CPU disponibles au sein de la machine virtuelle.

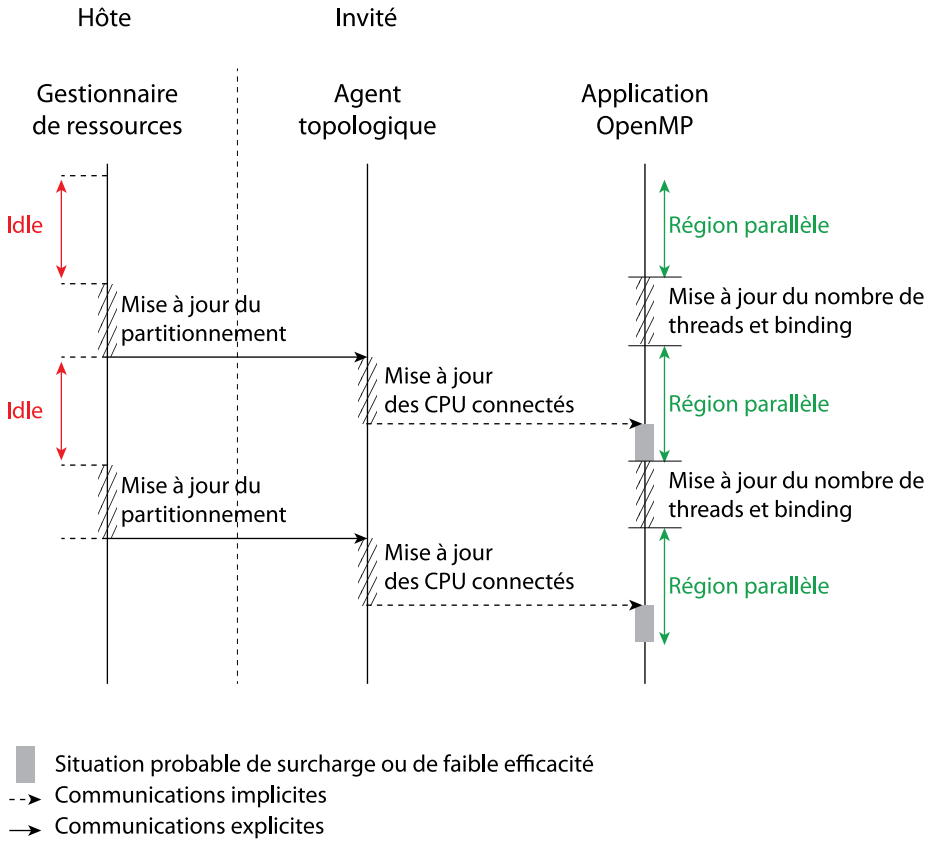


FIGURE 2.15 – Mécanisme de partitionnement

> The thread that encountered the parallel construct becomes the master thread of the new team, with a thread number of zero for the duration of the new parallel region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that parallel region.

FIGURE 2.16 – Extrait norme OpenMP — parallel construct (section 2.5)

Le changement du nombre de threads ne pouvant être réalisé qu'entre deux régions parallèles, il est possible d'ajouter un appel de fonction entre le début d'une région parallèle et son initialisation. Pour cela, nous avons besoin de connaitre les fonctions réalisant le début et la fin d'une région parallèle.

Limitations Notre approche apporte deux contraintes pour qu'un code OpenMP s'exécute convenablement. La première est d'ordre technique et conduit à une exécution incohérente du problème. En effet, les variables utilisées par un code OpenMP peuvent être partagées (shared) ou privées (private), dans le cas des variables partagées tous les threads utilisent la même variable. À l'inverse quand une donnée est privée alors chaque thread garde une copie qui lui est propre. Cette variable est stockée dans un espace mémoire spécifique dénommé TLS (Thread Local Storage) dans la mémoire du thread, une variable privée conserve son statut d'une région parallèle à une autre. La suppression d'un thread entre deux régions parallèles sans prendre en compte la présence de TLS risque d'entrainer une exécution erronée. La figure 2.17 et les deux exemples de codes OpenMP associés illustrent le cas de la gestion d'un faux partage entre threads. Pour s'exécuter en concurrence, les threads ont besoin d'un degré de liberté maximale, des accès mémoire à une même page mémoire peuvent être problématiques pour certaines applications. Le résultat du code sans faux partage entraine une perte de portion de tableau ou une erreur de segmentation selon que l'on augmente ou diminue le nombre de threads utilisés dans une région parallèle. Cette première limite est toutefois à nuancer. Il est en effet conseillé de ne pas dimensionner un problème avec le nombre de threads. La deuxième contrainte n'entraine pas de bug, mais entrave le mécanisme mis en place par notre algorithme, en effet si le code possède un nombre très réduit de régions parallèles alors nous ne pourrons ajouter ou retirer de threads. En pratique, les codes de simulation possèdent soit une région parallèle par phase de calcul soit des régions imbriquées. Notre approche reste donc suffisamment générique pour la plupart des codes OpenMP.

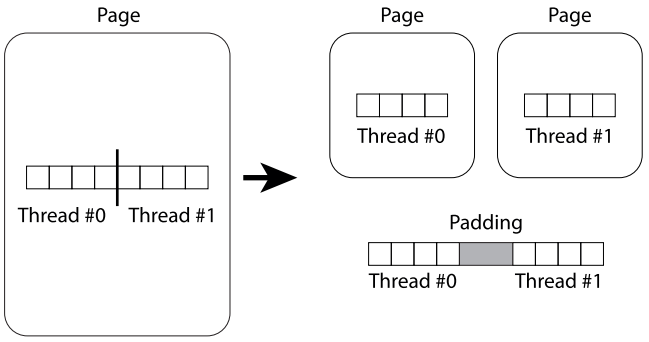


FIGURE 2.17 – Illustration d'un partage de tableau entre threads OpenMP

```
1  int tab_global[NB_ELTS];
2
3  #pragma omp parallel for
4  {
5    for(i = 0; i < NB_ELTS; i++)
6      fn(tab_global);
7  }
```

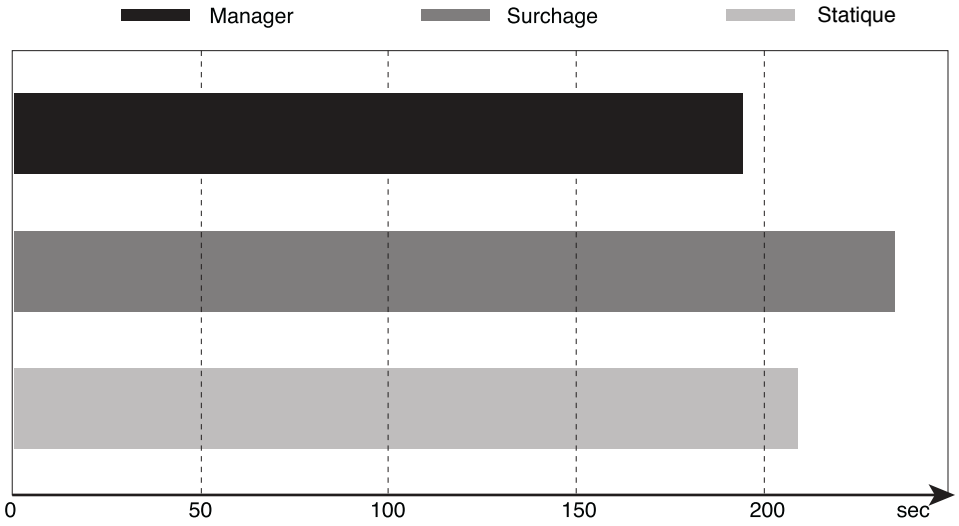
Utilisation sans risque de faux partage

```
1  omp_set_num_threads(4)
2
3  #pragma omp parallel private(tab_local)
4  {
5    thread_id = omp_get_thread_num();
6    first_rank_in_tab = (nelts+PADDING) * thread_id;
7    tab_local = tab_global[first_rank_in_tab];
8  }
9
10 omp_set_num_threads(2)
11
12 #pragma omp parallel private(tab_local)
13 {
14   for(i = 0; i < nelts; i++)
15     fn(tab_local);
16 }
```

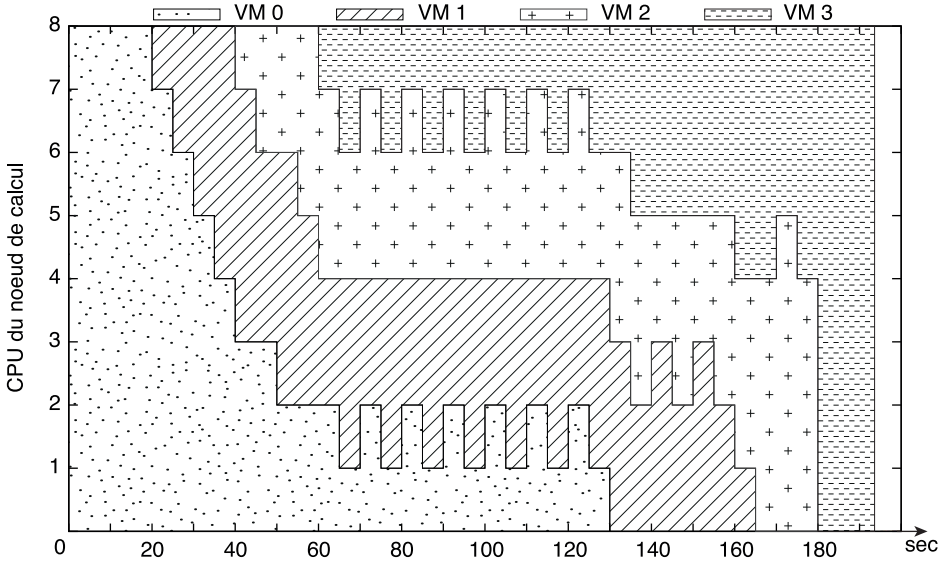
2.6 Evaluation de performances

Le premier test de performance que nous réalisons consiste à exécuter quatre fois un même benchmark avec un décalage de 20 secondes. Pour cela, on a fixé trois cadres d'exécutions que sont : la surcharge, le partitionnement statique et le manager. La surcharge consiste à exécuter les quatre benchmarks sans environnement virtuel de manière indépendante et concurrente, les CPU se retrouvent donc en situation de surcharge. Le partitionnement statique exécute les quatre benchmarks sans environnement virtuel en anticipant les prochaines exécutions, chaque benchmark s'exécute donc avec deux cœurs. Enfin le manager répartit dynamiquement les CPU du nœud de calcul entre les machines virtuelles en fonction de leur activité.

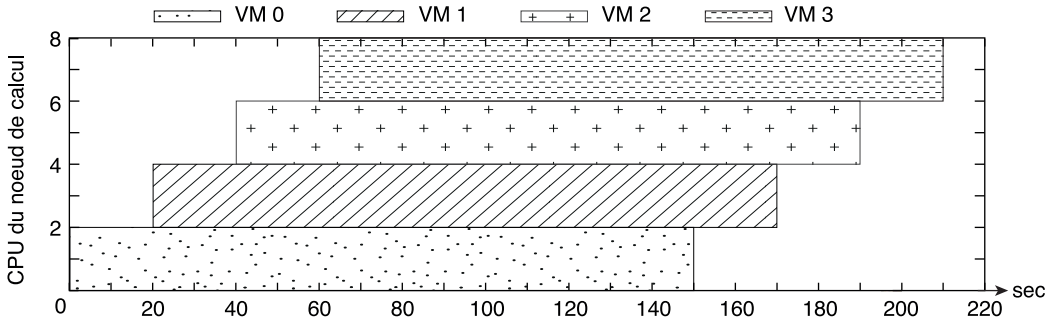
La figure 2.18a présente les temps d'exécution de chacune des trois situations. On observe une fois encore que la situation de surcharge est à éviter même dans le cas natif car elle présente un trop grand surcoût en terme de temps d'exécution. On remarque aussi que l'exécution au sein de machines virtuelles et l'utilisation d'un manager permet de gagner en temps d'exécution total dans quatre benchmarks. L'utilisation du manager permet d'exploiter plus efficacement les ressources de calcul vis-à-vis du mode statique, la figure 2.18 présente l'attribution des CPU à chaque machine virtuelle au court du temps. On constate que les huit cœurs du nœud de calcul sont utilisés pendant l'intégralité du benchmark ce qui n'est pas le cas du mode statique (figure 2.18c). Le partitionnement est mis à jour toutes les cinq secondes sur la base d'une prise de mesure des flops de chaque VCPU. Les benchmarks étant identiques on s'attend à une répartition homogène des CPU, c'est globalement le cas à un cœur près. On peut expliquer cette fluctuation par deux remarques. La première est le décalage entre la mise à jour des CPU attribués à une machine virtuelle et sa prise en compte par le runtime OpenMP. La seconde tient dans le fonctionnement de notre algorithme. En effet, c'est la diminution du ratio flops/nombre de CPU lors de l'acquisition d'un nouveau CPU qui permet



(a) Temps d'exécution en fonction de la politique choisie



(b) Variation du nombre cœurs aux fonctions du temps pour la politique manager



(c) Variation du nombre cœurs aux fonctions du temps pour la politique statique

FIGURE 2.18 – Execution de 4 benchmarks NAS Open de classe B

de confirmer cette attribution lors du prochain partitionnement. On a donc besoin de prendre un cœur en plus pour savoir s'il améliorera nos performances, on observe cette affirmation lors des oscillations du nombre de cœurs d'une machine virtuelle.

Le deuxième test vise à montrer que l'exécution en concurrence de plusieurs benchmarks peut être plus intéressante que leur exécution en série. En effet, on sait qu'une application ne présente pas une accélération linéaire en fonction du nombre de threads utilisées. L'efficacité globale du code diminue à mesure que l'on augmente le nombre de threads utilisés par l'application. Cette observation conduit à l'hypothèse suivante : deux exécutions concurrentes d'un même programme sur un sous-ensemble de CPU peuvent être plus rapides que leur exécution en série sur tous les CPU de la machine. Évidemment, l'exécution en concurrence aura un impact sur l'utilisation des caches et les débits mémoires, mais ce surcoût n'est pas toujours prépondérant. Nous avons considéré sur la figure 2.19 l'exécution du NAS BT pour une, deux et quatre exécutions du benchmark en concurrence. On compare le temps d'exécution en série natif avec le temps d'exécution en concurrence virtuelle (orchestré par notre gestionnaire de ressources). On remarque que notre gestionnaire présente un surcoût pour l'exécution d'un benchmark. Celui-ci est dû à la latence de notre gestionnaire à détecter et assigner des cœurs à une machine virtuelle en sommeil. Pour les autres cas, on remarque que notre gestionnaire arrive à améliorer légèrement le temps d'exécution en série des benchmarks. Cette remarque est d'autant plus vraie que l'on n'observe plus aucun surcoût à la virtualisation sur l'exécution complète des benchmarks. Ce surcoût étant absorbé par l'ordonnancement de ressources.

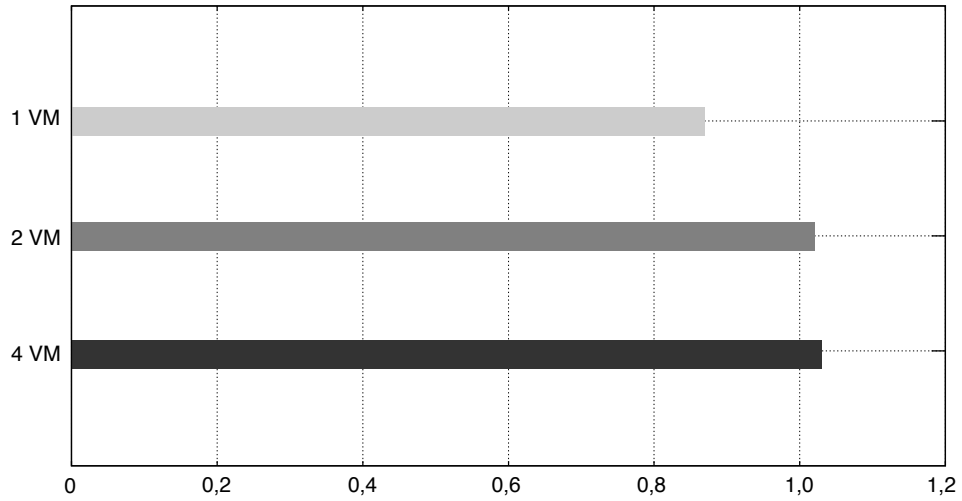


FIGURE 2.19 – Accélération d'une exécution en parallèle vis-à-vis d'une exécution en série

Conclusion

Dans ce chapitre, nous avons mesuré le surcoût en temps d'exécution au sein d'une machine virtuelle d'une application multithreadée. Nous avons ensuite étudié le fonctionnement de l'hyperviseur QEMU afin d'identifier et optimiser les mécanismes responsables de ce surcoût. Pour cela, nous avons ajouté une prise en compte du caractère NUMA d'une topologie pour une machine virtuelle. Cette modification a permis une diminution conséquente du surcoût observé initialement, le rendant ainsi raisonnable pour une utilisation HPC. Ce résultat était important, car il valide notre proposition

d'utiliser des machines virtuelles pour exécuter des codes patrimoniaux et permettre de fournir des environnements personnalisés à destination des utilisateurs de cluster HPC.

Sur la base de l'utilisation de machines virtuelles, nous avons pensé et développé un gestionnaire de ressources CPU pour nœud de calcul. Le principe repose sur l'ajout et la suppression dynamique de CPU exposés au sein d'une machine virtuelle. Afin de rendre pertinente cette gestion dynamique, nous avons considéré l'utilisation des CPU par une machine virtuelle en mesurant le nombre d'opérations flottantes réalisées par les VCPU des machines virtuelles s'exécutant en concurrence. Les machines utilisant le plus intensément les CPU qui lui sont confiés se voient confier plus de CPU qu'une machine sollicitant peu ses CPU.

Enfin, nous avons illustré l'exécution de gestionnaire de ressources pour un cadre concret : la négociation transparente de ressources entre applications OpenMP concurrentes. Les résultats ont montré qu'il est possible de mesurer l'activité d'une application OpenMP ainsi que redimensionner son nombre de threads à la volée. L'accélération d'une application n'étant pas linéaire, plus on augmente le nombre de threads plus l'accélération par thread diminue. Nos résultats montrent qu'il est possible d'améliorer le temps d'exécution global d'une série d'applications en l'ordonnançant pour une exécution en concurrence sans surcharge vis-à-vis d'une exécution en série de ces mêmes applications.

Afin d'étendre l'utilisation de processus dynamiques aux programmes MPI, nous allons maintenant nous intéresser aux communications réseau réalisables par une machine virtuelle et leurs caractéristiques en terme de latence, débit et facilité de migration.

Chapitre 3

Développement d'un support exécutif MPI VM-aware

"Dans la communication, le plus compliqué n'est ni le message, ni la technique, mais le recepteur"

DOMINIQUE WOLTON

Dans ce chapitre, nous nous intéressons à l'usage du réseau par une machine virtuelle préservant une migration sans contrainte d'une machine virtuelle dans le cadre d'applications de calcul intensif sur supercalculateur. Nous rappellerons les solutions existantes et utilisées dans des architectures de type informatique en nuage. Cette étude nous a conduits à mettre en évidence les contraintes principales induites par le contexte de calcul haute performance et qui empêchent une utilisation directe des outils de type cloud dans une architecture HPC. Nous analyserons ensuite le fonctionnement d'un support exécutif MPI afin de comprendre comment un message est traité depuis le programme utilisateur jusqu'au traitement des communications réseau. Nous avons choisi pour cette étude le fonctionnement de MPC[PJN08]. Nous étudierons plus particulièrement les mécanismes d'initialisation et de routage d'un message par le support exécutif MPC. MPC nous permettra de mettre en place l'accès à un réseau pour une machine virtuelle. Pour finir, nous détaillerons l'architecture d'un support exécutif adapté aux machines virtuelles (VM-aware) et la création de connexions entre applications MPI lancées dans des machines virtuelles différentes.

3.1 Support exécutif MPI pour machine virtuelle

3.1.1 Migration et communications réseau

La migration consiste à déplacer une machine virtuelle, ce déplacement peut se faire en local sur un même nœud de calcul ou en distant vers un nœud de calcul différent. On parle de migration à froid lorsque la machine virtuelle est éteinte avant d'être déplacée ou de migration à chaud quand l'opération est réalisée en temps réel. Dans notre cas, c'est la migration à chaud qui nous intéresse, en effet cette migration n'interrompt pas les programmes ou services qu'elle héberge. À l'aide de la migration, on souhaite pouvoir déplacer une machine virtuelle pour ajuster son placement et permettre ainsi une diminution du temps d'envoi et de réception des messages avec ses voisins. La migration peut aussi permettre de suspendre l'activité d'un nœud pour des raisons telles que la prévention d'une probable panne ou la réalisation de tâche d'administration (mise à jour, maintenance ...).

Comme nous l'avons présenté dans l'introduction (cf. chapitre1), il existe plusieurs techniques permettant l'accès direct ou émulé à une ressource réseau pour une machine virtuelle. Parmi celles-ci, seuls PCI-Passthrough et SR-IOV permettent d'obtenir une latence raisonnable pour envisager des communications entre machines virtuelles sur deux nœuds de calcul. On rappelle que le PCI-passthrough consiste à attribuer une ressource physique à une machine virtuelle. Ce mécanisme supprime l'accès au périphérique pour le système d'exploitation hôte. L'absence de cet accès empêche l'exécution d'un job concurrent au sein du nœud de calcul sur lequel s'exécute la machine virtuelle. Cette contrainte empêche l'usage de PCI-Passthrough sur les nœuds de calcul d'un supercalculateur. SR-IOV quant à lui ne supprime pas l'accès au périphérique pour l'hôte, mais fournit une vision virtuelle du périphérique pour la machine virtuelle. Son fonctionnement induit lors de la migration des contraintes trop fortes pour être utilisées de manière générique. En effet, comme le montrent des travaux de prototypage d'un mécanisme de migration SR-IOV, il est nécessaire de sauvegarder la valeur de la fonction virtuelle (VF) de l'hôte pendant la migration et de reprendre la même valeur une fois la machine virtuelle migrée. Le processus échoue donc si la VF du périphérique présent sur le nœud destinataire est déjà utilisée. En plus des contraintes inhérentes à SR-IOV et PCI-Passtrhough, il s'ajoute un problème de la compatibilité. Le PCI-Passthrough et SR-IOV nécessitent une forte compatibilité entre le BIOS, le système d'exploitation et le matériel. Ces solutions satisfont parfaitement les besoins en performance du HPC, mais elles ne permettent pas de conserver de manière robuste le mécanisme de migration d'une machine virtuelle.

Dans les outils de gestion de cluster de type cloud, la migration est assurée par des instances réparties sur l'ensemble des nœuds de calcul formant le cluster. Ces instances ont pour but de contrôler l'accès aux différentes ressources d'un nœud de calcul par une ou plusieurs machines virtuelles. Cela rend la machine virtuelle agnostique du réseau et des ressources physiques, en effet c'est l'instance hôte qui joue le rôle de ressources physiques pour la machine physique. La migration est ainsi facilitée par la présence d'un gestionnaire d'instances virtuelles et d'une instance permettant à la machine virtuelle de se lancer ou de migrer vers une autre. Dans le cas de Vsphere, on trouve les instances hôtes ESX et un gestionnaire multi hôtes Vcenter qui pilote la migration. Dans notre cas, le supercalculateur n'est pas dédié à un cluster de machine virtuelle. Nous avons donc besoin de mettre en place une instance dynamique de gestion de machine virtuelle côté hôte afin de permettre le déplacement d'une machine virtuelle vers des nœuds nouvellement alloués.

Nous avons montré les différentes solutions possibles pour réaliser des communications réseau entre plusieurs machines virtuelles et leurs limites. Partant de ce constat, nous proposons dans ce chapitre une solution permettant d'une part d'obtenir une faible latence et d'autre part de conserver la facilité de migration d'une machine virtuelle.

3.1.2 Protocole de message MPI de bout à bout

L'utilisation des fonctionnalités des périphériques physiques d'une machine repose sur des protocoles et pilotes réseau. Les fonctionnalités sont appelées par le biais d'une interface de programmation (API), chaque périphérique ou technologie possédant en général une API qui lui est propre. Une API permet de décrire les actions réalisables par le matériel ou le pilote associé, elle consiste en un ensemble normalisé de classes, de méthodes et de fonctions. Les API peuvent être spécifiques à un type de technologie par exemple l'API verbs pour les cartes Infiniband. Mais aussi générique comme c'est le cas pour CCI qui permet d'utiliser un grand nombre de technologies réseau (Ether-

net, Infiniband, Gemini, SeaStar ...) à l'aide d'une même API. L'utilisation d'une API spécifique est bien souvent complexe et demande une connaissance approfondie de la technologie utilisée, elle permet d'optimiser l'utilisation d'une ressource de manière très fine. Un module spécifique doit être développé et maintenu pour chaque API que l'on souhaite supporter, le gain d'optimisation a donc un coût en temps de développement et en veille technologique pour les maintenir à jour. À l'inverse, une API générique ne permet pas un comportement propre à chaque matériel, mais offre un fort degré de portabilité.

Dans notre cas, nous avons choisi de nous baser sur un protocole de message de type MPI. Les supports exécutifs MPI étant optimisés pour prendre en charge les différents réseaux utilisés au sein des supercalculateurs, ce choix offre donc une portabilité maximum pour une solution HPC. À cela s'ajoute le fait que le routage de messages qu'il soit entre les processus MPI de l'utilisateur ou les machines virtuelles qui les contiennent sera entièrement de type MPI. Ce choix a donc l'avantage de nous permettre de développer un protocole de messages unifiés aussi bien du point de vue des communications hôte vers invité qu'entre invités ou hôtes. Il sera donc possible de réduire une partie des traitements communs aux deux niveaux de routage.

Nous rappelons que MPI est un standard de programmation à mémoire distribuée basé sur des synchronisations explicites par le biais de messages. L'envoi de ces messages peut être réalisé soit par le biais d'une mémoire partagée entre deux processus soit par une interface réseau. Nous avons vu qu'il est nécessaire pour des raisons de migration de machine virtuelle de fournir une couche de réseau indépendante du réseau sous-jacent. L'ajout de cette couche réseau crée un empilement de protocoles réseau, un premier au niveau de l'invité dans lequel s'exécute notre processus MPI et un deuxième au niveau de l'hôte lors de l'envoi d'un message entre deux machines virtuelles. Cela a pour conséquence d'alourdir le coût du traitement d'un message entre machines virtuelles, une encapsulation et un acquittement pour chaque niveau de la couche réseau. Cette remarque nous a conduits à l'implémentation d'un support exécutif adapté à l'exécution de processus MPI au sein de machines virtuelles.

3.1.3 Présentation du support exécutif MPC

3.1.3.1 Architecture globale

MPC est un support exécutif MPI basé sur des processus légers (*threads*) nommés tâches MPI. Les données de ces tâches MPI sont privatisées afin de reproduire le comportement classique des processus UNIX. Cette privatisation permet l'utilisation de MPC sans modification du code utilisateur, pour autant les tâches MPI d'un même processus MPC partagent le même espace d'adressage. Ce partage permet l'échange de message sans recopie (*zéro copy*) entre tâches MPI d'un même processus MPC. De plus, le partage de l'espace d'adressage entre tâches MPC permet au support exécutif de réduire son empreinte mémoire. Cette mutualisation de l'espace mémoire entre les tâches MPC permet de réaliser du recouvrement au travers du *collaborative polling*. L'idée étant de permettre aux tâches inactives ou en attente actives de traiter les messages destinés à une autre tâche du processus MPC. L'utilisation de tâches MPI et de leur ordonnanceur est recommandée pour profiter des optimisations du support exécutif et donc améliorer les performances d'un code de calcul grâce à MPC.

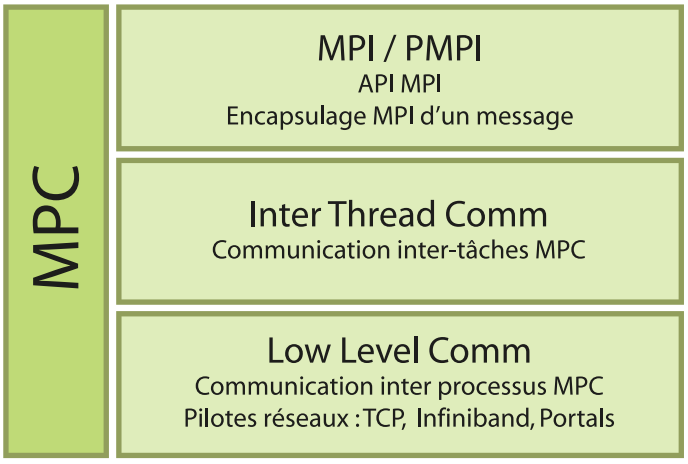


FIGURE 3.1 – Architecture globale MPC

3.1.3.2 Pilote au sein de MPC

Les pilotes du *Low-Level-Comm* sont organisés par rails de communication. Chaque rail est indépendant et possède ses propres routes statiques et dynamiques. Les routes sont dites statiques si elles sont construites dès l'initialisation et ne peuvent être détruites ou dynamiques si elles sont créées à la demande en fonction des communications qu'effectue le programme MPI. De plus, il existe un système de filtre appelé *gateway* qui permet de refuser un message et de l'envoyer vers un autre rail. Ces filtres peuvent être implicites, c'est-à-dire que le rail possède ou non une route vers le destinataire du message ou explicite par le biais d'une fonction fournie au rail. Le routage d'un message est obtenu en parcourant les rails dans l'ordre décroissant des priorités jusqu'à trouver le rail de plus grande priorité dont le filtre ne rejette pas le message. La dernière caractéristique d'un rail est de pouvoir construire à la demande ou non une route vers un destinataire en passant par un ensemble de messages de contrôle nécessaire à l'initialisation des structures d'envoi et réception de message.

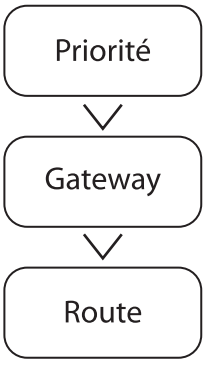


FIGURE 3.2 – Choix d'un rail MPC

3.2 Implémentation d'un pilote SHM de communication entre processus

Comme nous l'avons présenté, MPC implémente des tâches MPI sous forme de processus léger. Cela entraine de facto une utilisation de mémoire partagée entre les tâches MPC d'un même processus MPC. Dans son mode nominal purement MPI, MPC lance un processus par nœud de calcul puis une tâche par cœur de chacun des nœuds. Cela

explique l'absence de pilote SHM dans les précédentes versions de MPC, son fonctionnement par défaut n'impliquant pas de communication entre processus d'un même nœud. Cette remarque est toutefois à nuancer, en effet, le mode tâche de MPC implique que les variables globales aux processus d'un programme MPI puissent être privatisées pendant la compilation. Pour cela, MPC fournit un support de privatisation automatique soit par le biais du compilateur GNU (gcc) présent dans MPC soit par le biais d'une option de compilation pour le compilateur Intel (icc). Pour autant, ce support de privatisation automatique ne gère pas tous les types de variables globales. C'est le cas des variables globales dont l'initialisation est dynamique. Notre implémentation de SHM est donc motivée par le besoin de proposer une exécution de MPC purement basée sur des processus. En outre, l'implémentation de notre module de communication entre machine hôte et machine invité repose sur l'utilisation d'une mémoire partagée. Le module SHM ainsi développé couvre donc tous nos besoins.

Le module SHM peut être aussi un moyen de réduire la contention entre les tâches MPC en les regroupant dans des équipes différentes (processus) tout en profitant de la latence et de la bande passante de la mémoire partagée d'un nœud de calcul. Notre module doit donc présenter un haut niveau de robustesse et performance afin d'être intégré dans la version distribuée de MPC.

3.2.1 Échange de messages entre processus en mémoire partagée

La mémoire peut être partagée à l'aide de différentes techniques plus ou moins adaptées aux contraintes de performance liées au calcul haute performance. Nous nous intéresserons en priorité à deux mécanismes que sont le recours à un module noyau et la projection de pages physiques communes au sein de l'espace d'adressage des processus.

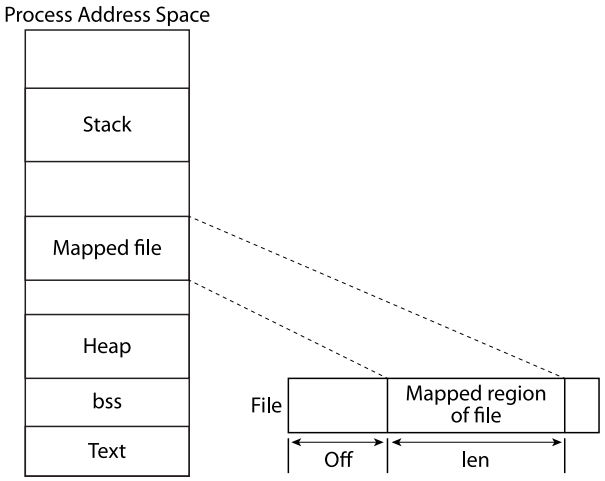


FIGURE 3.3 – Fonctionnement d'un mmap

3.2.1.1 Segment de mémoire partagée

L'approche classique consiste à utiliser un descripteur de fichier afin de permettre à plusieurs processus de partager un même espace de mémoire physique. Les pages associées à cette mémoire physique sont ensuite projetées dans l'espace d'adressage virtuel de chacun des processus. Les processus peuvent ensuite écrire et lire les données écrites sur ces pages en y accédant à l'aide d'un segment contigu d'adresses virtuelles du processus. Dans un système UNIX, on utilise les fonctions `shm_open` pour ouvrir un fichier de la taille voulue, on force ensuite la création d'une correspondance entre le fichier ouvert

et un ensemble de pages physiques à l'aide de la fonction `troncate`. Enfin l'utilisation de `mmap` permet de projeter l'espace physique pointé par le descripteur de fichier au sein de l'espace d'adressage virtuel d'un processus. Il est à noter que l'usage de mmap échappe à l'allocateur de la bibliothèque C, l'utilisateur s'étant directement alloué la zone mémoire. La zone mémoire échappe alors au mécanisme propre à l'allocateur malloc et doit être gérée de manière statique. De plus, l'adresse de début de ce segment est propre à chaque processus, en effet l'opération de projection dans l'espace d'adressage du processus consiste à trouver un segment libre. Le résultat de cette recherche est donc souvent lié aux allocations précédemment réalisées par le processus. L'utilisation d'adresses mémoires réelles n'est donc pas a priori possible dans un segment de mémoire partagée inter processus pour cette raison. Pour lever cette contrainte, l'approche classique consiste à utiliser un mécanisme d'adresses virtuelles, c'est à dire d'utiliser une traduction pour que les adresses placées dans le segment de mémoire partagée soient cohérentes pour tous les processus.

L'adressage relatif et absolu quant à lui repose sur la nature contiguë de l'adressage virtuel du segment de mémoire partagée. Pour cela, on considère que le segment de mémoire partagée commence à l'adresse 0. Les adresses relatives qui sont calculées sont donc la différence entre l'adresse virtuelle connue par un processus et l'adresse de début du segment pour ce même processus. Chaque processus peut ensuite recalculer l'adresse réelle en y ajoutant l'adresse du début du segment. Cette méthode permet de modifier dynamiquement les adresses des processus ayant accès au segment partagé. Il est par contre obligatoire de traduire toutes les adresses faisant référence au segment lors d'un accès ou d'une écriture. On citera la bibliothèque OpenPA (Open Primitive Atomics) qui permet de gérer la nature asymétrique des accès au segment pour un ensemble de processus. La file implémentée est à producteur multiple et consommateur unique. OpenPA permet de gérer un seul morceau de mémoire partagée, la valeur de l'adresse de début du segment de mémoire partagée d'un processus étant stockée sous la forme d'une variable globale.

Nous avons vu que l'utilisation de l'allocateur «malloc» et donc d'allocation dynamique n'est pas possible de manière transparente au sein d'un segment de mémoire partagée, pour autant l'utilisateur peut vouloir utiliser son propre allocateur de mémoire. Dans le cas de MPC, l'allocateur permet de construire des chaines d'allocation créées à la demande de l'utilisateur et qui prennent en argument un pointeur de début et une taille ainsi que le support de thread désiré (l'utilisateur précise ainsi si la chaine a besoin d'être threadsafe ou non). La solution de l'adressage relatif ne permet pas son utilisation au sein d'un segment de mémoire partagée, car il faudrait qu'il traduise en interne les adresses qu'il manipule. Pour lever cette contrainte, nous avons implémenté avec Sébatien Valat l'outil `MPC-SHM-mapper` permettant de créer un segment de mémoire partagée tel que tous les processus possèdent le même segment d'adresse virtuelle pour cette région. Pour cela, nous avons besoin de connaître le nombre de processus qui utilisent le segment de mémoire partagée. L'implémentation est réalisée dans un mode maître-esclave, le maître est chargé de choisir et d'envoyer aux esclaves l'adresse de début du segment de mémoire partagée. Si un processus ne partage pas la même adresse que celle du maître alors les processus envoient les segments libres de leur espace d'adressage virtuel. Le maître réalise une intersection des espaces libres afin d'élire la prochaine adresse de début du segment de mémoire partagée. Si tous les processus partagent la même adresse, l'élection est réussie et l'adresse commune trouvée. Les mécanismes de synchronisation entre les processus ont été implémentés à l'aide des fonctionnalités de PMI et de MPI. Son utilisation peut se faire de manière indépendante de MPC, il repose sur deux fonctions permettant de créer et supprimer un segment de

mémoire partagée.

Notre implémentation du module SHM peut donc reposer sur une gestion statique d'un segment de mémoire partagée, les adresses sont traduites à chaque accès ou écriture d'une donnée au sein du segment par un processus. Elle peut également reposer sur une gestion dynamique en utilisant la synergie de `MPC-SHM-mapper` et de l'allocateur interne de MPC.

3.2.1.2 Utilisation d'un module noyau

L'utilisation de mécanismes s'exécutant en mode privilégié permet de réaliser des échanges de message *zéro copie*, c'est-à-dire que le message est copié directement dans la mémoire du processus distant sans aucune copie intermédiaire. Cela permet de réduire la latence pour les messages de moyenne ou grande taille, en effet les petits messages ont un temps de recopie négligeable devant le temps de traitement du message. Ce mécanisme peut être réalisé par le biais des nouvelles fonctionnalités du noyau Linux 3.2 ou par le biais de module noyau comme KNEM. Pour des raisons de contrainte de configuration des clusters à notre disposition, nous avons testé uniquement les CMA (Cross Memory Attach) qui ont été ajoutés au sein des noyaux Linux 3.10+. Les CMA permettent de réaliser des lectures et des écritures dans la mémoire virtuelle d'un processus distant à condition que le programme possède la capacité `SYS_CAP_PTRACE` ou que les deux processus appartiennent au même groupe et au même utilisateur.

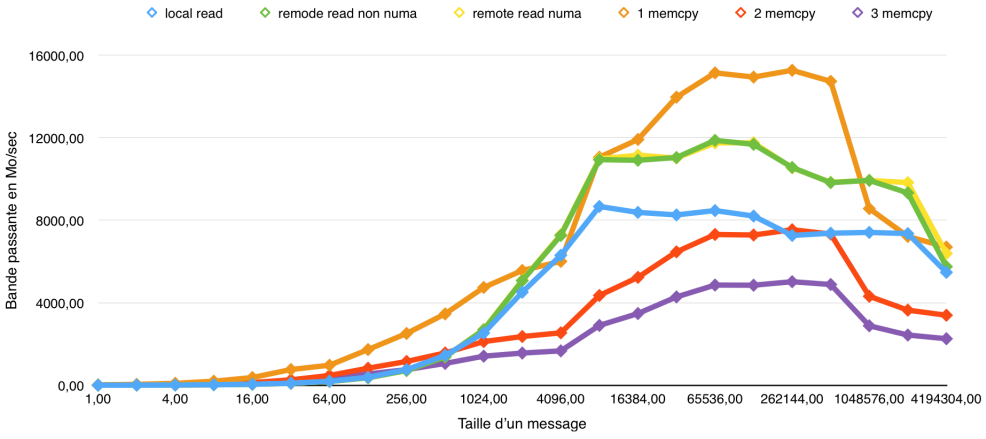


FIGURE 3.4 – Temps de copie d'un message pour les fonctions memcpy et process_vm_read (CMA)

Nous avons donc comparé les débits moyens d'un, deux et trois memcpy ainsi que ceux d'un CMA d'un processus vers lui même et d'un CMA entre processus distincts. Les processus peuvent alors être localisés sur le même nœud NUMA ou sur deux nœuds différents. Le graphe nous permet d'observer le surcoût lié à la traduction des pages virtuelles en comparant les courbes d'un memcpy et celles de copies CMA. On peut observer que les CMA ne sont pas appropriés dans le cas de petits messages même comparativement à une approche impliquant trois copies. Notre graphe permet de montrer l'intérêt que l'usage de CMA est pertinente dans le cas de message d'une taille supérieure à 1024 octets. Cette remarque est à nuancer, car le test réalisé ne reflète pas l'exact comportement des CMA dans un module SHM. En effet, il ne prend pas en compte le surcoût induit par le protocole de rendez-vous permettant l'échange des adresses entre deux processus. De surcroit, l'envoyeur doit attendre l'acquittement de la lecture réalisée par le processus qui reçoit.

Les CMA nécessitent un échange d'information pour être réalisés, le processus qui lit ou écrit doit connaître l'adresse de la donnée à copier au sein du processus distant, mais aussi prévenir celui-ci quand il a fini de réaliser son action. L'usage des CMA ne présente pas de bonne performance pour de petits messages, les CMA ne semblent donc pas être un bon support pour les messages de contrôle propres à réaliser un CMA, mais aussi pour les petits messages.

3.2.2 Files de messages et routage d'un message

Dans le mode SHM, chaque processus réserve son propre segment et le partage avec les autres processus présents sur le même nœud de calcul. Le choix d'un segment par processus est motivé par le fait que cela nous assure par construction que les segments sont positionnés sur le bon nœud NUMA et qu'ils puissent être migrés au besoin. Ce choix est aussi dû à une utilisation future de SHM comme module de communication hôte/invité si MPC est lancé dans une machine virtuelle, les segments pourront alors être ajoutés et supprimés au gré des migrations sans influer sur les autres processus du nœud. La figure ci-dessous présente l'organisation d'un segment de mémoire partagée pour un processus MPC.

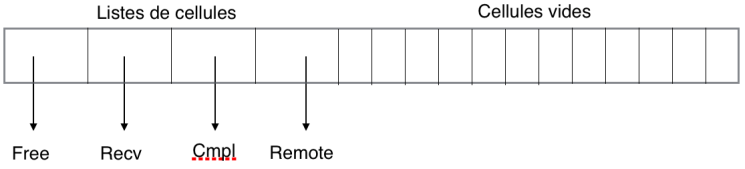


FIGURE 3.5 – Organisation d'un segment de mémoire partagée

Lors de la phase d'initialisation du module SHM, chaque processus initialise son segment en plaçant de manière statique les structures de file partagée (Free, Recv, Cmpl et remote). Le reste du segment est découpé en cellules de taille identique. Chaque cellule possède un entête permettant de les ajouter à une des quatre files du segment. Les files de messages sont pour le moment des files avec verrous supportant un adressage relatif afin de n'avoir aucune contrainte sur la projection du segment au sein de l'espace d'adressage de plusieurs processus. Toutes les cellules sont ensuite ajoutées à la file de cellules libres (Free). Les autres processus s'allouent un tableau de pointeurs afin de sauvegarder l'adresse absolue des files de messages de leurs voisins. La configuration du module SHM (nombre et taille des cellules) peut être modifiée par l'utilisateur par le biais du fichier de configuration de MPC, l'espace utilisé par le module SHM est donc modulable en fonction des besoins. Par défaut, chaque processus possède 32 cellules de 8ko. Actuellement, le module SHM peut fonctionner dans le cas le plus minimaliste avec deux files (Free et Recv) et une cellule de la taille d'un entête MPC de message inter processus. Notre implémentation s'inspire du pilote Nemesis implémenté dans MPICH.

L'initialisation des routes se fait de manière statique sur une topologie complète entre tous les processus d'un même nœud. En effet, il n'y a pas d'intérêt a priori à restreindre l'accès à un segment de mémoire partagée d'un processus. La mémoire utilisée restant identique, quel que soit le nombre de processus accédant au segment de mémoire partagée. Construire une route SHM consiste dans la version actuelle de SHM à associer le rang global d'un processus MPI à son rang local sur le nœud. Le rang local permet de retrouver dans le tableau les pointeurs de files de l'ensemble des processus du nœud, les files du processeur auquel on envoie un message. Les routes permettent d'envoyer des

messages uniquement au sein d'un nœud de calcul, SHM ne peut donc pas être utilisé indépendamment d'un rail permettant des communications inter nœud. L'utilisation du rail SHM peut se faire avec un rail Infiniband, TCP dans la version actuelle de MPC. La figure 3.6 illustre l'envoi d'un message à destination d'un processus. Lorsque le rail SHM ne possède pas de route vers le processus destinataire, le message est envoyé au rail IB et TCP. Le rail SHM n'ayant que des routes vers des processus locaux, nous nous sommes parfaitement intégrés au fonctionnement nominal de MPC.

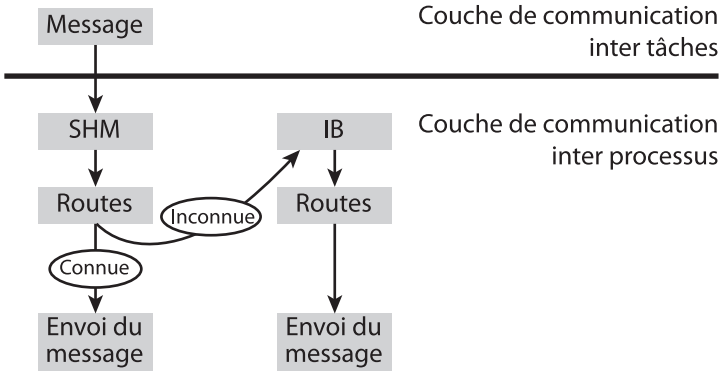


FIGURE 3.6 – Routage d'un message vers le rail de communication

3.2.3 Envoi et réception d'un message en mémoire partagée

3.2.3.1 Gestion des messages de la taille d'une cellule

L'envoi d'un message est détaillé dans la figure 3.8 et peut être découpé en 4 étapes.

1. Réservation d'une cellule inutilisée
2. Copie du message et ajout à la file des messages à envoyer
3. Distribution des messages à envoyer dans les files de réception
4. Copie du message et libération de la cellule.

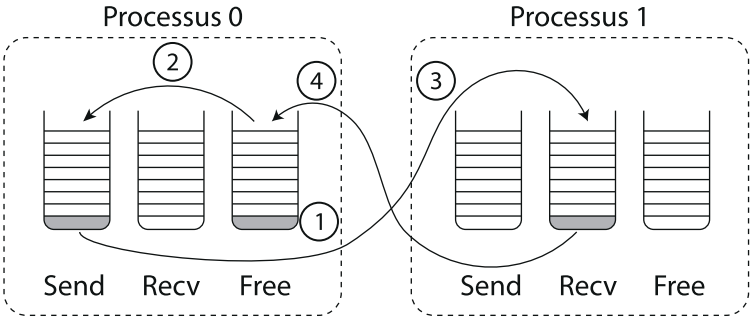


FIGURE 3.7 – Envoi d'un message SHM

L'utilisation d'une file d'envois pour les processus permet de garder un comportement de producteur unique et consommateur multiple, cette caractérisation permettant l'utilisation de files lockfree comme celles d'OpenPA. Quand un processus ne trouve pas le message attendu dans sa file de réception, il va chercher les les messages présents dans la file d'envois du processus dont il attend le message. Il fait donc avancer les messages pour tous les autres processus jusqu'à trouver le sien. L'étape 3 est facultative dans le cas de message ANY_SOURCE ou si le message récupéré est le message attendu par le processus réalisant le traitement des messages à envoyer.

Afin de pouvoir gérer plusieurs morceaux de mémoire partagée, nous avons dû supprimer dans un premier temps le mécanisme de la file SEND au profit d'un ajout directement dans la file de messages de RECV du processus distant. En effet, lors de l'usage de plusieurs segments partagés il est nécessaire de savoir d'où provient le message. Les cellules d'un segment ne peuvent alors pas être assignées aux files d'un autre segment. La traduction de son adresse virtuelle donnerait une adresse incohérente, le processus ne pouvant connaître à l'avance d'où provient la cellule reçue, il ne pourra lire correctement ses informations.

Les cellules de messages ont une taille limitée et ne peuvent contenir que des messages dont la taille de l'entête et des données ne dépasse pas la taille de la cellule. Il nous faut donc mettre en place un mécanisme dédié au traitement de messages plus grands qu'une cellule. Pour cela, nous avons implémenté un mécanisme à base de messages fragmentés ou d'appel CMA. Dans les deux cas, les cellules seront utilisées pour échanger les données ou informations nécessaires au protocole d'échange d'un gros message.

3.2.3.2 Gestion des gros messages avec support CMA

Pour optimiser le nombre de copies réalisées en mode CMA et EAGER, on a ajouté un mode sans recopie qui consiste à ne pas recopier la cellule dans la mémoire du processus. On envoie directement la cellule vers la couche de communication entre tâches MPC avec les fonctions de libération et copie adaptée. Les modes avec copie et sans copie sont configurables par le biais de l'interface XML de configuration de MPC. Attention dans le cas d'une configuration avec peu de cellules le mode sans recopie pourrait dans la version actuelle de SHM entrainer une situation de famine de cellule et donc des situation de bloquages (*deadlock*).

3.2.3.3 Gestion des gros messages sans support CMA

Le support CMA ne permet pas une portabilité maximale pour notre pilote SHM, en effet comme nous l'avons présenté précédemment ils ne sont disponibles que dans les noyaux 3.10+. Ils peuvent être bloqués par des mécanismes de sécurité rendant leur utilisation impossible même avec le support CMA intégré au noyau. Le service YAMA, lorsqu'il est activé, empêche les appels CMA pour éviter qu'un malware puisse lire la mémoire d'un processus. De plus, les appels CMA entrainent le punaisage et la traduction de pages virtuelles en pages physiques. Après la lecture ou l'écriture des pages distantes, les pages sont dépunaisées et la traduction perdue. L'algorithme mis en place est couteux et ne profite pas d'effet de mémoire vis-à-vis des traductions précédentes. Il faut donc être sûr que le mécanisme de rendez-vous CMA et la lecture distante n'entrainent pas un coût supérieur à un envoi par fragment de message. Un pilote SHM doit donc intégrer un mécanisme de gestion d'un envoi par message fragment comme illustré sur la figure 3.8.

Dans l'exemple de la figure 3.8, le message est de la taille de cinq cellules et trois segments peuvent être envoyés simultanément. Nous avons limité le nombre de cellules participant à un envoi de fragment afin d'éviter une famille de cellule qui peut entrainer une synchronisation entre les processus lorsque des messages de tailles différents sont envoyés de manière croisée entre des processus distincts. L'absence de cellules génère une situation d'attente pour les processus émetteurs, quelque soit le type de messages.

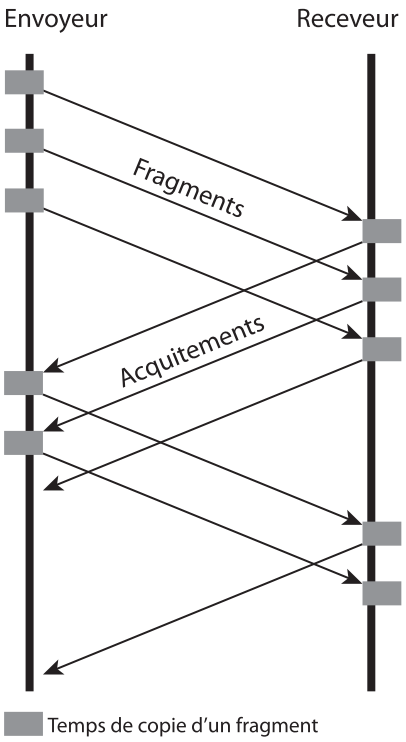


FIGURE 3.8 – Envoi d'un message SHM

3.2.4 Évaluation de performances

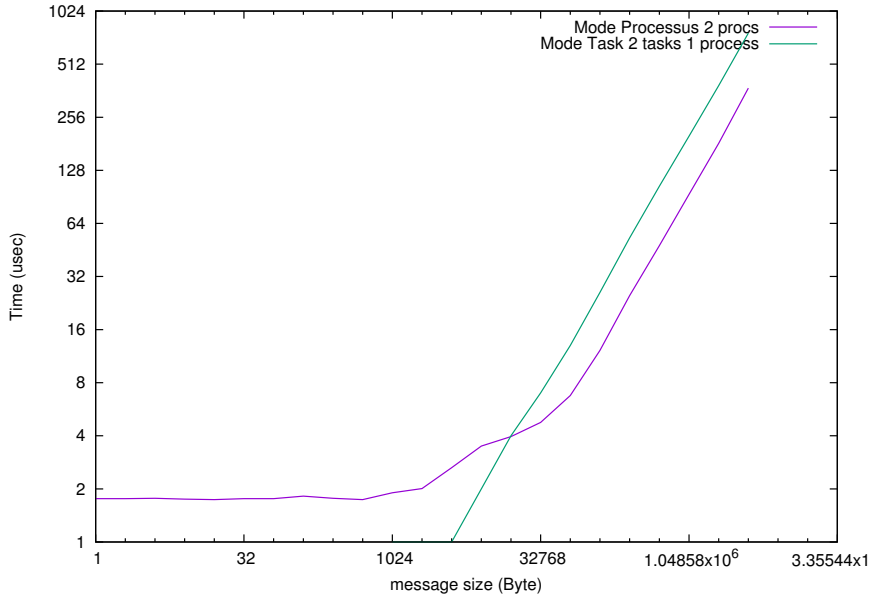
Nous avons comparé nos résultats avec l'implémentation des envois de messages entre tâches de MPC, nous n'avons pas comparé nos résultats à d'autres implémentations. La comparaison d'un support exécutif avec support MPI_THREAD_MULTIPLE avec une implémentation MPI thread based sans ce support ne serait pas fair-play. Notre module se veut une étape pour réaliser des communications entre machines virtuelles, afin de rivaliser en performance d'autres supports exécutifs il devra faire l'objet d'optimisation.

3.3 Extension du module SHM aux communications entre machines virtuelles

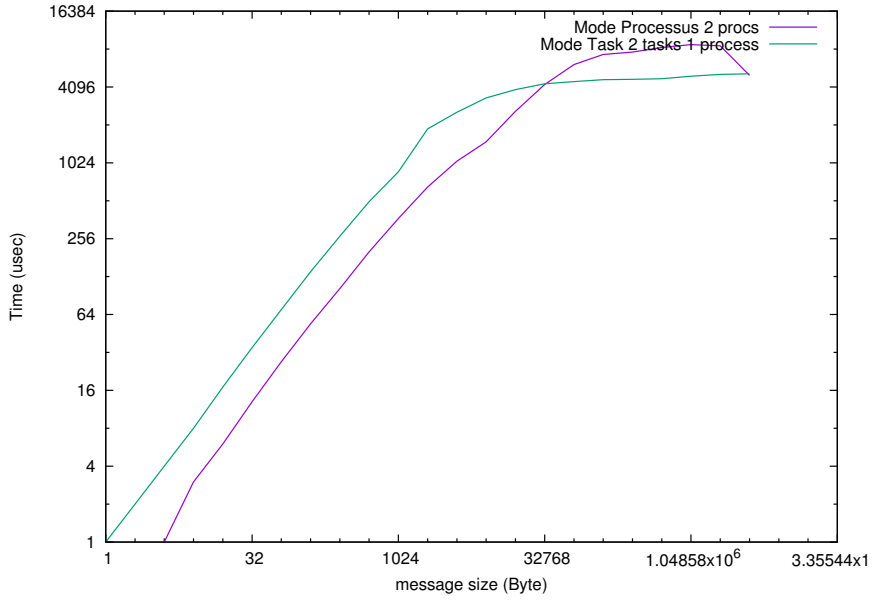
Dans notre description de l'architecture générale de notre outil, nous avons présenté le module SHM comme le meilleur candidat pour être l'interface entre l'espace invité et l'espace hôte. Nous avons montré dans la partie précédente que notre implémentation est performante et robuste en la confrontant à la suite de benchmark Intel. Nous allons maintenant présenter des modifications propres à son utilisation au sein de notre solution virtualisée.

3.3.1 Mémoire partagée entre machines hôtes et invitées

Nous avons présenté dans notre introduction le périphérique virtuel IVSHMEM, on rappelle qu'IVSHMEM est implémenté dans QEMU et permet la création d'un segment de mémoire partagée entre l'hôte et l'invité. Ce segment peut être partagé entre plusieurs invités et permet donc de partager de la mémoire entre plusieurs invités. L'allocation de l'espace partagé coté hôte se fait de la même manière classique, on crée un descripteur de fichier auquel on associe une zone mémoire. La zone mémoire est ensuite mappée dans le processus de l'hôte, cette dernière étape n'est pas possible au sein de l'invité. Pour l'invité, il est possible coté hôte de fournir un descripteur de fichier pointant vers une



(a) IMB-Pingpong



(b) IMB-Bcast

FIGURE 3.9 – Comparaison du mode Tâche de MPC avec le module SHM de MPC

zone mémoire partagée à QEMU afin qu'il puisse projeter l'espace mémoire alloué par un processus UNIX hôte. L'hyperviseur se charge ensuite d'exposer cet espace mémoire sous forme d'un périphérique PCI pour le noyau invité. On initialise le périphérique PCI, permettant l'accès au segment partagé, par le biais du module noyau fourni par IVSHMEM. Le module noyau est utilisé uniquement au sein de la machine virtuelle et ne constitue pas une limite au déploiement de notre solution. Nous avons donc dans un premier temps adapté le pilote SHM de MPC afin qu'il prenne en charge l'initialisation d'un périphérique PCI au sein de l'invité.

3.3.2 Routage de message et migration de machines virtuelles

Les machines virtuelles étant susceptibles de migrer, nous avons besoin d'un module SHM permettant de détruire ou créer à la demande une route vers les autres processus MPI. Nous avons donc implémenté au sein de MPC l'ensemble des contrôles permettant de construire une route à la demande.

3.3.3 Échange de message avec recopie entre machines virtuelles

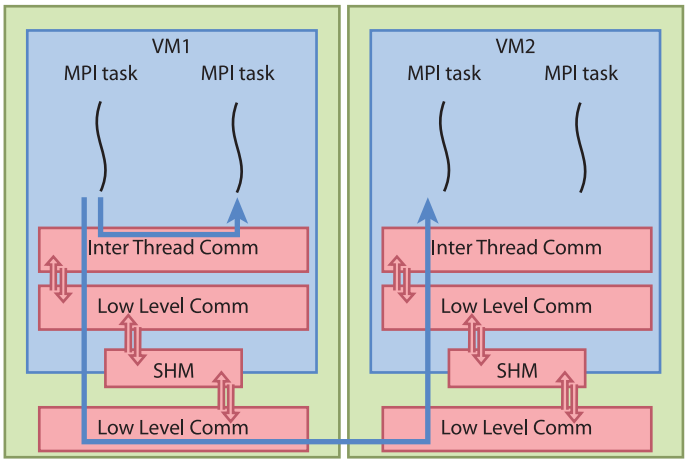


FIGURE 3.10 – Traitement d'un message MPI

Comme on peut le voir sur la figure 3.10, deux machines virtuelles sont lancées avec deux tâches MPC par machine virtuelle pour un total de 4 tâches MPI. On peut voir le routage à deux niveaux.

Pour sortir une information de la machine virtuelle, nous avons besoin de la rendre accessible pour l'hôte. Pour cela, nous utilisons conjointement IVSHMEM pour instancier une mémoire partagée entre invité et hôte. À partir de cette zone mémoire, les pilotes pour mémoire partagée des instances hôte et invité de MPC peuvent s'envoyer des messages. Côté invité, les messages vers une tâche MPC d'une autre machine virtuelle sont toujours envoyés vers le pilote SHM. Coté hôte, le message est routé vers le pilote SHM dans le cas des messages à destination de sa machine virtuelle ou vers un pilote réseau dans le cas contraire.

Nous avons montré qu'il est possible de réaliser des communications efficaces entre deux instances MPC. Cependant, l'utilisation d'une mémoire partagée implique une recopie complète des données à envoyer. Comme nous l'avons montré, le temps de recopie est négligeable pour des messages de petite taille. Il est donc essentiel de mettre en place

un mécanisme adapté spécifiquement aux gros messages à l'aide de protocole de type RDMA afin de pallier le temps de recopie dans la mémoire partagée.

Nous avons montré qu'il est possible d'initialiser une instance MPC au sein de chaque machine virtuelle créée par l'instance MPC sur l'hôte. Maintenant, nous allons détailler comment les messages sont routés entre les processus MPI invités.

3.3.4 Envoi de requêtes RDMA entre machines virtuelles

Un protocole RDMA permet à un processus directement d'écrire et lire les données d'un processus d'une machine distante (remote). Le RDMA permet principalement d'éviter une recopie d'un message comme c'est le cas pour TCP par exemple, mais aussi d'avoir une gestion asynchrone de message par le biais de buffer pré alloué et dont les adresses sont connues. On parle de *put* lorsque l'envoyeur écrit dans la mémoire du processus receveur et à l'inverse de *get* lorsque le processus receveur va lire dans la mémoire du processus envoyeur. Dans le cas de MPC, les messages sont à l'initiative de l'envoyeur et on réalise principalement des *get*.

3.3.4.1 Envoi d'une requête RDMA au sein de l'invité

Pour mettre en place un mécanisme de RDMA entre nos machines virtuelles, nous avons besoin de permettre à l'instance hôte de MPC de lire et d'écrire directement dans la mémoire virtuelle de la machine virtuelle.

Dans notre architecture, chaque processus MPC lance une machine virtuelle. QEMU et MPC hôtes partagent donc le même espace d'adressage de mémoire virtuelle. QEMU est en charge de l'allocation de la mémoire physique du système d'exploitation invité. MPC a donc par construction accès aux informations contenues dans l'espace d'adressage physique de la machine virtuelle. Cependant pour lire dans la mémoire d'un processus utilisateur au sein de la machine virtuelle, il faut avoir accès à son adressage virtuel. On a donc besoin de traduire les adresses virtuelles d'un processus invité pour permettre à MPC de réaliser du RDMA. Cette traduction permet d'obtenir un vecteur I/O, c'est-à-dire un vecteur d'adresse mémoire et d'offset permettant de connaître la localisation des données recherchées. En effet, la mémoire virtuelle est contiguë pour le processus utilisateur, mais discontinu du point de vue de l'adressage physique. Cette traduction est couteuse et n'est réalisée que pour de gros messages, dont le temps de recopie est trop couteux pour être réalisé à chaque envoi.

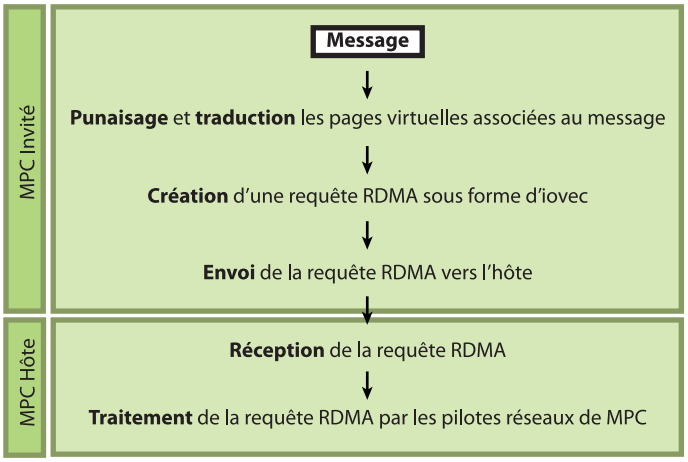


FIGURE 3.11 – Traitement d'une requête RDMA

Nous avons donc au sein de l'invité un module noyau permettant de traduire une adresse virtuelle en adresse physique, le vecteur ainsi obtenu est placé dans un message SHM à destination du MPC hôte. Cette opération doit être réalisée par les deux processus MPC participant à l'échange du message.

3.4 Lancement de MPC en configuration distribuée

Maintenant que nous avons mis en place une interface de communication entre deux machines virtuelles à l'aide d'un protocole en mémoire partagée, nous avons besoin de relier les processus MPC participant à l'exécution d'une même application. Ce mécanisme est, dans le cas classique d'une exécution MPI, géré par des services comme PMI ou Hydra. Nous décrivons dans cette section le déploiement natif de MPC dans un scénario mono et multi nœuds puis le déploiement distribué de MPC au sein de machines virtuelles localisées sur le même nœud de calcul, enfin nous étendrons au cas multi nœuds.

3.4.1 Initilisation d'un support exécutif MPI : PMI et Hydra

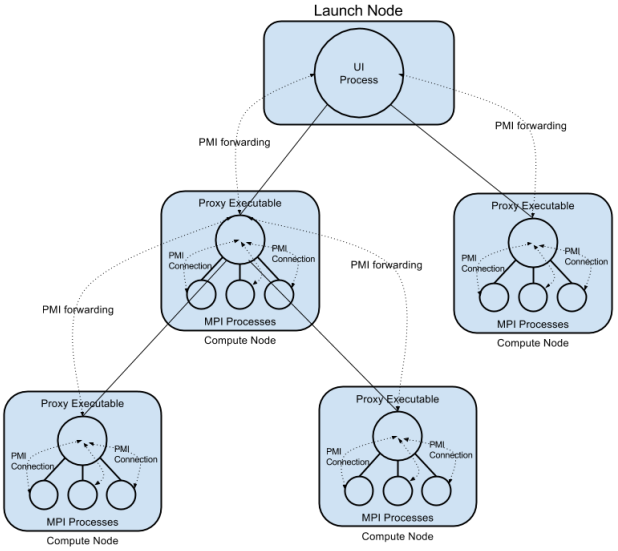


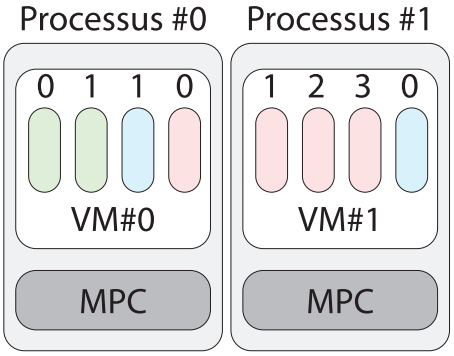
FIGURE 3.12 – Lancement d'un programme MPI avec Hydra (source Wikipédia)

Un support exécutif MPI a besoin de synchroniser des informations pour créer ses propres routes. Pour cela, les bibliothèques PMI et Hydra fournissent une API basée sur des requêtes vers une base de données clés valeurs. Seuls les processus d'une même allocation SLURM et lancés par SLURM peuvent interagir entre eux. Cette situation empêche l'utilisation directe de PMI pour nos processus invités. Nous avons donc pensé un algorithme pour permettre des synchronisations entre instances MPC invité et hôte à l'aide d'une découverte de route de type DNS. Cette solution se veut scalable et dynamique.

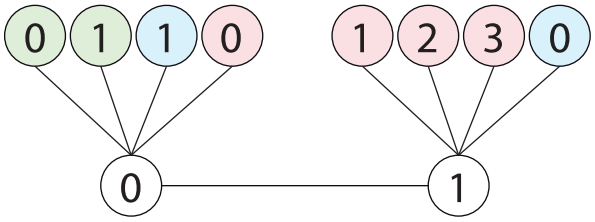
3.4.2 Routage d'un message entre machines virtuelles

Nous avons décrit dans la première section de ce chapitre ce que l'on utilise pour les communications MPI que ce soit pour les échanges entre machines virtuelles ou entre les processus internes à une machine virtuelle. Nous avons donc plusieurs numérotations :

la numérotation des machines virtuelles et les numérotations de processus MPI propres à chaque application. Comme on peut le voir sur la figure 3.13, les applications d'une même machine virtuelle peuvent avoir des numéros de processus différents ou identiques. Le numéro de processus MPI ne suffit pas à réaliser un routage de bout en bout vers un processus distant. Pour réaliser un routage de bout en bout on a besoin de la couleur du processus et d'un numéro de processus.



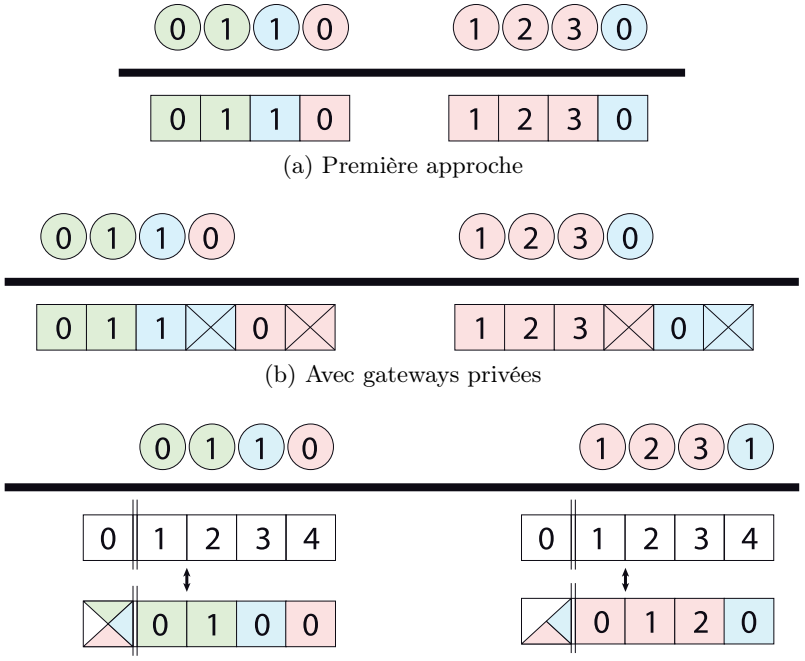
(a) Première approche



(b) Vue schématique

FIGURE 3.13 – Deux machines virtuelles exécutant des applications MPI

L'idée des couleurs et de numérotation relative ou locale rappelle la notion de communicateur MPI, pour cette raison nous avons décidé de considérer que chaque processus MPC hôte possède un communicateur regroupant tous les processus qui s'exécutent dans sa machine virtuelle. Chaque processus MPI d'une application MPI se voit donc attribuer un rang MPI unique au sein du communicateur de l'hôte ainsi qu'une couleur. Dans notre exemple précédent, le résultat est présenté dans la figure aux rangs locaux, on ajoute une gateway afin de pouvoir intercepter les messages placés dans une autre machine virtuelle. Un communicateur MPI n'étant pas extensible, nous considérerons que le support exécutif MPI s'exécutant sur l'hôte peut être lancé avec un argument spécialement ajouté pour notre cas *-pmax*. Ce nouvel argument permet de préciser au support exécutif MPI qu'un communicateur peut être plus grand que le nombre de processus nécessaires à l'application. Ce comportement reste cohérent avec la norme MPI en effet, rien n'empêche d'avoir un communicateur plus grand que le nombre de processus MPI d'une application. Dans le cadre de la tolérance aux pannes, la norme MPI introduit la notion de réduction de la taille d'un communicateur quand un processus est victime d'une erreur par exemple. Aux processus MPI utilisés par les applications s'ajoutent, pour des besoins de routages, des gateways vers les autres processus MPC hôte. Nous avons donc dimensionné le communicateur de chaque processus MPI hôte pour qu'il accepte un nombre de processus égal au nombre de cœurs disponibles dans la machine qu'il héberge. On ajoute ensuite un processus supplémentaire pour recevoir et envoyer des messages vers d'autres machines virtuelles.



(a) Première approche

(b) Avec gateways privées

(c) Avec gateways partagées

FIGURE 3.14 – Organisation du communicateur MPC hôte

Dans la figure 3.14b, on voit que les applications présentes dans plusieurs machines virtuelles possèdent une gateway symbolisée par une croix. Une gateway nous assure qu'un message entre machines virtuelles est échangé entre les instances MPC hôtes jusqu'à trouver l'hôte destinataire.

3.4.3 Serveur DNS pour applications MPI (DCS)

Nous avons expliqué dans la section précédente que l'on souhaitait attribuer à une application une couleur pour séparer le traitement des messages MPI provenant d'une machine virtuelle. Les processus invités sont lancés localement dans une machine virtuelle et doivent être regroupés avec les processus invités d'une autre machine virtuelle. Le routage d'un message repose sur l'utilisation de trois informations : un hôte, une couleur, un rang local. Il est nécessaire de donner la même couleur à tous les processus d'une même application. Pour cela, nous allons considérer que chaque application se voit attribuer une clé unique lors de son lancement. Cette clé va permettre aux processus invités de trouver quelle couleur leur est affectée. Ce comportement est similaire à celui d'un DNS qui est capable de donner une adresse IP en fonction d'un nom de domaine.

3.4.3.1 DNS centralisé

Nous allons supposer dans cette partie que nous avons un gestionnaire de tâches capable d'attribuer à une application MPI des ressources au sein d'un cluster virtuel. Les communications entre machines virtuelles étant couteuses, nous souhaitons limiter le rôle de ce job manager uniquement à l'attribution des ressources et non à la mise en place des synchronisations entre applications MPI de machines virtuelles différentes. Pour les synchronisations, nous allons considérer que s'exécute indépendamment du gestionnaire de tâches, un serveur DNS centralisé qui enregistre le lancement d'un job à l'aide d'une clé et qui va lui attribuer une couleur.

Nous avons illustré le mécanisme de découverte des autres processus au sein du cluster virtuel dans deux situations : deux processus, une machine virtuelle et deux processus, deux machines virtuelles. Dans les deux cas, le résultat final doit nous permettre d'attribuer un rang global à chaque rang MPI invité ainsi que nous permettre de router des messages entre les machines virtuelles. Dans les figures 3.15 et 3.16, les informations en rouge sont les informations ajoutées lors de l'envoi d'une requête à l'inverse, le bleu définit les informations mises à jour par un acquittement d'une requête. La numérotation représente un ordre global des envois de requête pour chaque application MPC ayant lancé des processus MPI au sein d'un invité. De plus, les informations en noir sont considérées comme connues dès le lancement de l'application.

Lors du lancement d'une application MPI, chaque processus d'une même application à l'intérieur d'une machine virtuelle connait ses voisins. En effet, lancer une application MPI locale à une machine virtuelle est exactement pareil que de lancer une application MPI sur un nœud de calcul. Aucune connexion avec des nœuds distants n'est nécessaire et les processus MPI peuvent communiquer à l'aide d'une mémoire partagée. Ce comportement est représenté par les flèches 0 et 5 de la figure 3.15, un processus parmi les processus invités d'une application est élu et communique avec son hôte pour demander sa couleur et ses rangs MPI globaux (requête 1). Le nœud enregistre alors la clé ainsi que le nombre de processus locaux à sa machine virtuelle, il ajoute à la requête de l'invité son rang de processus MPI (ici 0). Comme il ne connait pas la couleur de K, il route la requête vers le serveur DNS (requête 2). Le DNS enregistre les informations de la requête dans sa table et complète la requête avec la couleur liée à la clé de l'application. La requête est ensuite renvoyée de proche en proche vers les processus invités ayant demandé l'information (3,4,5). Les rangs MPI locaux savent qu'ils sont sur le premier nœud partagé par l'application, il s'attribue donc les rang MPI globaux à l'application 0 et 1.

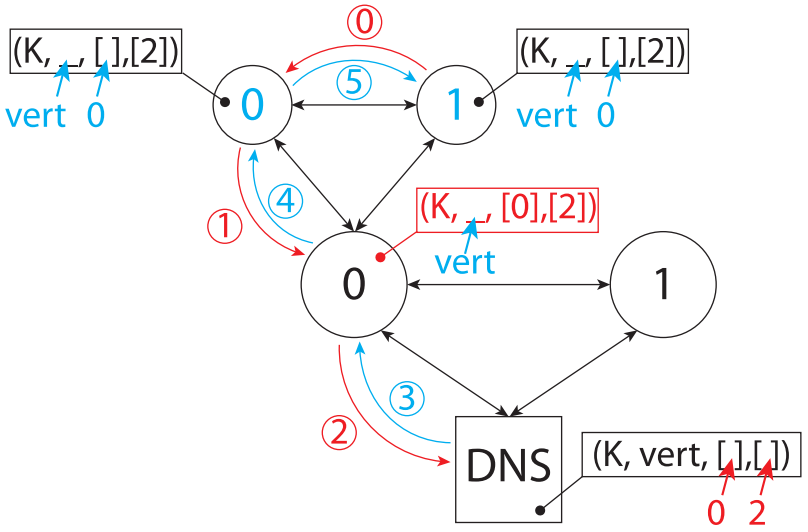


FIGURE 3.15 – Initialisation des routes pour deux processus et 1 machine virtuelle

Les messages de l'application seront par la suite routés par SHM s'ils sont échangés entre 0 et 1 ou envoyés vers l'hôte dans le cas où une route SHM interne à l'invité n'est pas trouvée. Pour l'hôte, les processus de sa machine virtuelle possèdent donc l'adresse (0, Vert, 0) et (0, Vert, 1). Ce cas nous a permis d'illustrer simplement l'exécution de résolution de sa couleur par une application, nous allons montrer le cas où l'application est lancée dans plusieurs machines virtuelles avec la figure 3.16.

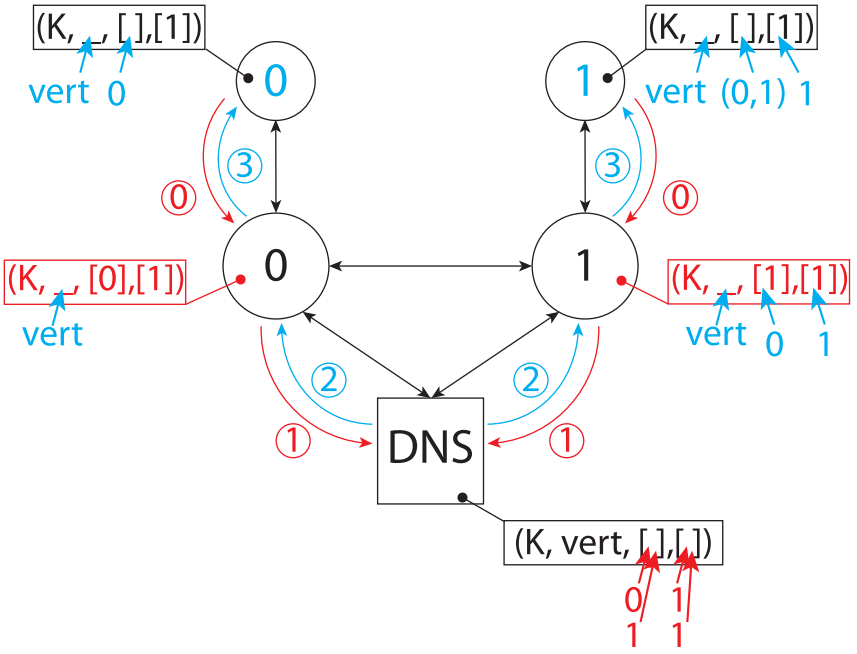


FIGURE 3.16 – Initialisation des routes pour deux processus et 2 machines virtuelles

Les numéros de requête et d'acquittement sont locaux à chaque processus, les actions pouvant être réalisées en concurrence pour les différents processus MPI hôte. Chaque processus étant seul dans sa machine virtuelle, il n'a pas de synchronisation locale à sa machine virtuelle et communique directement avec son processus hôte. Le serveur DNS reçoit deux requêtes d'enregistrement qu'il exécute dans l'ordre d'arrivée. Dans notre exemple, c'est la requête 1 de l'hôte 0 qui est arrivée en premier, le serveur DNS renvoie donc à 0 sa couleur qui est ensuite propagée vers le processus invité. Le processus invité voit qu'il est le premier a s'être enregistré car il n'y a qu'un seul numéro de nœud dans sa requête et s'attribue le rang MPI global 0. De la même manière, la requête de l'hôte 1 est acquittée en renvoyant la liste totale des nœuds précédemment enregistrés. Il est le deuxième nœud enregistré et le nœud précédent possède un seul processus, son rang MPI global est donc 1.

Les messages de l'application seront par la suite directement envoyés vers le processus hôte (gateway) pour envoyer un message vers un voisin. Pour l'hôte 0, la seule route connue est celle vers l'adresse (0,Vert,0) de son processus local. À l'inverse, l'hôte 1 connait les routes vers (0,Vert,0) et (1,Vert,0) découvertes pendant la résolution de couleur. Les messages du processus MPI 1 de l'application vers le processus MPI 0 de l'application seront donc directement envoyés vers (0,Vert,0). Dans le cas de l'hôte 0, une requête DNS sera nécessaire pour traduire le rang MPI de l'application en son adresse de routage (1,Vert,0).

On peut mélanger ensuite les deux mécanismes de résolutions de couleur afin de couvrir tous les cas de figure, la synchronisation des processus invités d'une même application réalisant une première synchronisation avec plusieurs machines virtuelles. La solution proposée présente un inconvénient majeur dans le cas de son exécution au sein d'un environnement HPC, elle est centralisée et donc ne passera pas à l'échelle. Nous avons donc considéré différentes topologies afin de construire une version distribuée de notre serveur DNS de cluster virtuel.

3.4.3.2 DNS et PMI

Dans le cas de PMI, nous avons vu que l'information est partagée par le biais d'une clé. Pour être scalable et topologique, on a besoin de savoir dans quel contexte une information locale est pertinente, autrement dit, dans quelle condition une donnée peut être localement invalidée. Dans le cas de PMI, la cohérence des données locales est assurée par le caractère non mutable d'une donnée échangée. Ce comportement empêche d'appliquer notre algorithme de serveur DNS avec l'aide des fonctions de l'API PMI, pour autant on ne peut pas se passer de l'usage de PMI pour le lancement et la mise en place des hôtes de machine virtuelle au lancement de notre cluster virtuel. Notre serveur DNS va nous permettre de découvrir les routes entre les processus invités lors du lancement d'un programme, mais aussi d'invalider une route comme c'est le cas lors d'une migration pour recréer dynamiquement la route vers les processus de la machine virtuelle migrée.

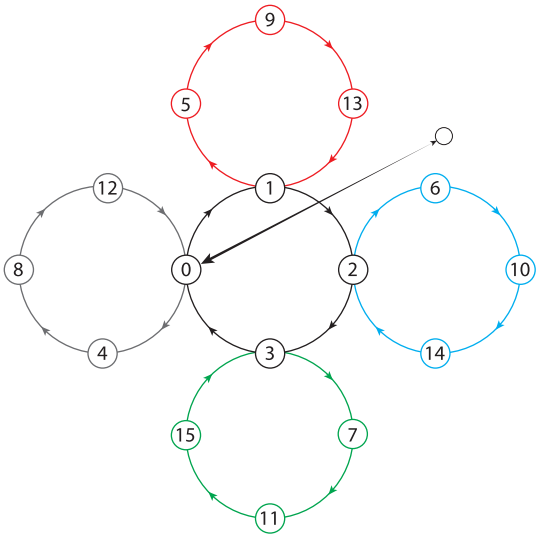


FIGURE 3.17 – Routage des requêtes DNS

Dans notre cas, le serveur DNS est un processus MPI hôte appartenant au sous-communicateur des processus MPI DNS. Comme on peur le voir sur la figure 3.17, les processus hôte MPI d'un même nœud de calcul sont reliés en anneau, de plus un processus par nœud est élu pour faire passerelle DNS.

Conclusion

Dans ce chapitre, nous avons étudié les moyens de communication disponibles pour une machine virtuelle. Nous avons ensuite montré leurs limites, c'est-à-dire leur impact sur les performances réseau et le mécanisme de migration d'une machine virtuelle. Les mécanismes de SR-IOV et de PCI-Passthrough permettent l'obtention de performances similaires à celles d'une utilisation native. Pour autant, leur usage implique une adhérence plus forte entre l'invité et l'hôte au travers de l'accès aux périphériques réseau. Ce constat nous a incités à considérer d'autres solutions pour les communications réseau de machine virtuelles dédiées au HPC.

Afin de répondre au besoin de migration d'une machine virtuelle sans contrainte matérielle, nous avons choisi de développer un support d'exécution MPI capable de s'adapter au fonctionnement d'une machine virtuelle. Ce travail est basé sur les concepts

et techniques définis par François Diakhaté pendant sa thèse. Notre solution se base sur l'utilisation de MPC qui est un support d'exécution MPI développé par le CEA. Notre idée est de découper MPC en deux parties : une partie communication à l'intérieur de la machine virtuelle et une partie à l'extérieur. Ce découpage a pour but de factoriser, côté hôtes et invités, les traitements inhérents à l'encapsulation et au routage d'un message MPI. Pour cela, nous avons étudié le fonctionnement interne de MPC et tout particulièrement la couche de communication et de synchronisation au démarrage de MPC entre processus d'une même application .

Afin de permettre l'envoi de message entre des processus MPI, nous avons implémenté un pilote SHM au sein de MPC. Ce pilote couplé à l'utilisation du périphérique virtuel IVSHMEM, nous permet une utilisation dynamique d'une interface de communication entre des instances MPC hôtes et invités. Cette interface est à même de supporter le mécanisme de migration et la création de route dynamique entre instance MPC hôte. Ce routage est transparent pour les processus invités au sein de la machine virtuelle, ainsi tout le mécanisme est intégré à la couche de communication entre processus de MPC.

Enfin, nous avons proposé un service DNS pour les processus MPC hôte. Ce service nous permet de connecter des instances MPC invitées entre elles par le biais d'un enregistrement auprès d'un serveur clé valeur distribué. Nous pensons que ce mécanisme est apte à répondre à nos besoins et permettra la réalisation de déploiement d'application MPI au sein d'un cluster de machines virtuelles. Nous n'avons pas pu par manque de temps produire une solution logicielle fonctionnelle pour les échanges de message entre processus MPI invités. L'implémentation et la stabilisation de l'initialisation d'un support exécutif entrainent un temps de développement complexe et long. Nous aurons la possibilité dans les mois à venir de vérifier que nos hypothèses sont réalisables.

Conclusion et perspectives

"L'avenir à chaque instant presse le présent d'être un souvenir."

Louis Aragon, Hourra l'Oural (1934)

Afin de répondre aux besoins croissants de la simulation numérique, les supercalculateurs sont devenus d'immenses machines dotées de plusieurs centaines de milliers de processeurs. Au fil des années, les grappes de calcul ont pris le pas sur les autres types d'architectures parallèles et constituent désormais le modèle dominant au sein des centres de calcul. Ces grappes sont structurées de manière hiérarchique. Des cœurs de calcul à la machine toute entière en passant par les nœuds, on ne distingue pas moins de six niveaux hiérarchiques pour une machine comme le supercalculateur Curie du Très Grand Centre de Calcul (TGCC). La réalisation d'applications ayant un fort degré de parallélisme, ainsi que l'estimation de leurs besoins réels en terme de ressources de calcul deviennent des défis pour les développeurs.

Pour optimiser et donc augmenter le degré de parallélisme d'une application, les développeurs ont à disposition des outils d'analyse comme TAU, SCALASCA ou encore MALP. Ces outils leur fournissent des informations sur les segments de code qui limitent le degré de parallélisme de leur application et donc la capacité de passage à l'échelle. Dans la pratique, les simulations ne sont pas des programmes réguliers. Elles alternent des phases de calcul dont certaines nécessitent l'accès à des ressources plus importantes. En règle générale, la réservation de ces ressources est statique en ce sens qu'il faut décider auparavant de la quantité nécessaire à l'exécution de l'application. C'est ainsi que fonctionnent les gestionnaires d'allocation comme TORQUE ou SLURM. Le dilemme se pose alors entre une allocation adaptée aux phases plus gourmandes en termes de ressources de calcul, quitte à ne pas utiliser des coeurs pendant certaines phases, et une allocation plus restreinte améliorant le taux d'utilisation des ressources mais augmentant également le temps d'exécution de l'application. De par cette problématique, il n'est pas rare de constater dans ces infrastructures une utilisation sous-optimale en moyenne des ressources de calcul d'une part, et la présence de ressources physiques isolées parmi des segments de ressources réservées d'autre part.

Enfin, un supercalculateur possède son propre environnement de programmation (compilateur, interpréteur, bibliothèque de calcul...). Une nouvelle machine de calcul implique un travail d'adaptation pour les applications. Dans la pratique, chaque changement d'environnement de programmation peut entrainer, même au sein d'une même machine, une modification du code ou de la compilation d'une application. Ces modifications demandent du temps et des développeurs ou utilisateurs capables d'isoler et de résoudre les problèmes induits par la migration d'une application. La pérennisation et la migration d'applications sont donc un enjeu économique important.

La gestion d'une grappe de calcul implique donc de s'attaquer à trois problématiques importantes que sont : l'irrégularité des besoins d'une application au fil de son exécution, la gestion statique des ressources et la pérennisation des applications HPC. Pour résoudre ces questions, la virtualisation peut être une première réponse. La virtualisation permet aux environnements de programmation d'être moins dépendants d'une architecture de machine particulière. De plus, l'abstraction de ressources inhérente au fonctionnement d'une machine virtuelle peut apporter une plus grande flexibilité dans le placement et l'affectation de ressources dédiées à l'exécution d'une application.

Contribution

Dans cette thèse, nous avons proposé un ensemble d'outils et de techniques afin de permettre l'utilisation de machines virtuelles dans un contexte HPC. L'objectif est de montrer que le surcoût de la virtualisation est raisonnable, c'est-à-dire de l'ordre de 2 à 3 %, afin de fournir des environnements de programmation personnalisés ou fixes aux développeurs d'applications. En effet, les machines virtuelles sont une première source de pérennisation d'un environnement de programmation. L'ensemble de la chaîne de compilation et d'exécution étant conservé dans la machine virtuelle, la migration d'une application vers une nouvelle machine est ainsi facilitée.

Tout au long de nos travaux, nous nous sommes efforcés de fournir et structurer notre contribution sur la base de la synergie d'applications ou services déjà existants et ce sans modifier leur fonctionnement en interne. Cette philosophie est basée sur l'idée qu'un outil en contexte de production doit être robuste. La robustesse ou fiabilité est une caractéristique essentielle d'un programme et demande une connaissance approfondie des concepts et techniques que l'on souhaite mettre en place. Dans le cadre d'un travail de thèse, nos outils et applications développés traitent et gèrent bien souvent des cas spécifiques limitant ainsi leur généralisation. De notre point de vue, il est important que les travaux réalisés soit exploitables pour une utilisation future. Le couplage d'outils développés par une large communauté active et structurée est un moyen d'y parvenir en fournissant un haut niveau de portabilité et de fiabilité. Pour autant, la synergie d'applications implique de se conformer aux mécanismes de chacun et reste une philosophie de développement complexe et longue.

Tout d'abord nous avons montré qu'il est possible d'optimiser le fonctionnement d'un hyperviseur afin de répondre le plus fidèlement aux contraintes du HPC que sont : le placement des fils d'exécution et la localité mémoire des données. Cette première étape nous a permis de valider la possibilité d'utiliser des machines virtuelles multithreadées pour des applications scientifiques. Pour la suite de nos travaux, nous avons développé une procédure de déploiement de machines virtuelles dédiées au HPC basées sur l'architecture d'un nœud de calcul obtenue par l'intermédiaire de l'outil HWLOC.

En s'appuyant sur ce lanceur de machines virtuelles, nous avons proposé un service de partitionnement des ressources d'un nœud de calcul entre machines virtuelles. L'objectif était de mettre en valeur les précédents résultats ainsi que de proposer un service capable d'absorber le surcoût résiduel induit par la virtualisation. Pour cela, nous avons mis en place une solution de mesure de performances des CPU virtuels afin de déterminer une efficacité d'utilisation des CPU physiques attribués à une machine virtuelle. Cette mesure est basée sur le nombre d'instructions flottantes par intervalle de temps. Ce mécanisme nous permet de moduler le nombre de CPU utilisables par une machine virtuelle ainsi que de prévenir toute situation de surcharge d'un CPU physique. La

variation de CPU visibles depuis l'espace utilisateur invité est un comportement problématique lors de l'exécution d'un programme multithreadé. Le fonctionnement des supports exécutifs ne lui permettant pas en général de prendre en compte cette variation pendant l'exécution du programme. Pour résoudre ce problème, nous avons ajouté des synchronisations implicites entre un support exécutif OpenMP et la modification des CPU connectés dans une machine virtuelle. Ces travaux nous ont permis de montrer qu'il est possible sous certaines conditions d'accélérer l'exécution totale d'une série de benchmarks. Ces benchmarks sont exécutés au sein de machines virtuelles différentes et leurs ressources sont réparties par notre gestionnaire de ressources.

À la suite de ces premiers travaux dont l'usage est réservé à des applications multithreadées, nous avons considéré l'exécution de code MPI au sein d'un cluster de machines virtuelles. Le cluster n'est ainsi plus limité à un nœud de calcul et son utilisation entraine des besoins de communications réseau entre les nœuds de calcul et donc entre les machines virtuelles. Afin de mettre en place une solution adaptée et apte à préserver une migration sans contrainte supplémentaire liée au matériel, nous avons décrit les solutions réseau pour machines virtuelles existantes ainsi que leurs limites. Aucune de ces solutions ne satisfaisant directement toutes nos contraintes, nous avons réalisé l'implémentation d'un système de communications spécifique. Celui-ci s'appuie sur un support d'exécution MPI capable de s'adapter à une exécution en environnement virtuel. Pour cela, nous sommes partis des concepts développés dans la thèse de François Diakhaté [DPNJ08] afin de coupler l'usage de l'hyperviseur QEMU avec le support exécutif MPI MPC.

L'adaptation du support exécutif MPC à une exécution VM-aware a nécessité l'implémentation d'un pilote SHM pour les communications entre processus MPI. Un travail d'adaptation a été nécessaire pour permettre une utilisation dynamique du pilote, c'est-à-dire son déploiement et sa destruction à la demande. Ainsi, il est possible de l'utiliser conjointement avec le périphérique virtuel IVSHMEM intégré à QEMU. Ce couplage dynamique nous assure que l'on peut échanger des données entre l'invité et l'hôte même quand le lien est temporairement détruit lors d'une migration. Nous avons ensuite réalisé un algorithme afin de lancer de manière distribuée des instances MPC invitées. Nous avons toutefois manqué de temps et n'avons pas pu réaliser une solution globale à l'exécution d'applications MPI au sein de machines virtuelles à l'issue de ces travaux.

Perspectives

Ces travaux ouvrent de nombreuses perspectives à plus ou moins long terme auxquelles nous allons contribuer à l'avenir.

Extension aux applications sensibles au débit mémoire

Dans un premier temps, nous envisageons d'étendre notre évaluation de l'efficacité d'une application afin de prendre en compte les accès mémoire réalisés par les différentes machines virtuelles. Pour le moment, nous nous basons uniquement sur le nombre d'instructions flottantes par seconde réalisées par les VCPU d'une machine virtuelle. Grâce à cette métrique, nous assignons des CPU à chacune des différentes applications scrutées afin de maximiser le débit d'instructions flottantes moyen réalisées par l'ensemble des CPU. Toutefois ce choix d'indice d'efficacité peut être remis en cause pour des applications sensibles aux débits mémoire. Ainsi la présence de deux applications partageant tout ou partie des liens mémoire (bancs NUMA, cache L3...) d'un nœud de calcul a pour effet de diviser la bande passante mémoire entre les applications. Pour garantir

une accélération de ce type d'application, il faudrait observer le débit mémoire d'une application en fonction du nombre de cœurs qui lui sont assignés. On pourrait alors identifier les caractéristiques de chaque application concurrente qu'elle soit limitée par le calcul (compute-bound) ou la mémoire (memory-bound). Il serait donc possible de placer sur les mêmes niveaux hiérarchiques des applications de natures différentes afin de réduire les interférences entre les applications.

Dans un second temps, il serait intéressant de s'attaquer au problème des applications réalisant des opérations entières. En effet, nous avons ciblé les applications à opérations flottantes pour isoler de facto le bruit système de nos mesures. Une partie du bruit système est due à l'exécution des services du noyau utilisant principalement des nombres entiers. Ainsi, nous pourrions étendre nos travaux à un plus grand nombre d'applications et notamment des applications de type Big-Data.

Placement et migration dynamiques de processus MPI

Aujourd'hui la plupart des applications sont écrites à l'aide du standard MPI pour s'exécuter au sein de plusieurs nœuds de calcul. Elles sont de plus en plus hybridées à l'aide d'un modèle de programmation à base de threads afin de tirer parti au mieux de la mémoire partagée d'un nœud de calcul. L'utilisateur se retrouve confronté au choix du nombre de processus par nœud de calcul ainsi que du nombre de threads par processus. Ce choix est statique et sera valable pour l'intégralité de l'exécution du programme. Pour éviter une sous-utilisation des ressources entre les threads et les processus, on ajoute aux applications des mécanismes d'équilibre de charge ou de vol de tâches. Ces mécanismes peuvent entrainer une plus forte contention dans le support exécutif à base de threads ou des échanges supplémentaires de messages pour le cas MPI. Enfin, l'équilibre de charge peut entrainer des pics d'utilisation mémoire notamment lors du repartitionnement d'un maillage entre les processus d'une application de type AMR. Grâce aux machines virtuelles dynamiques, les processus MPI peuvent être redimensionnés en nombre de threads en fonction de la charge d'un processus. De plus, les processus peuvent être migrés afin d'éviter que deux processus ayant une forte charge partagent le même nœud et soient en concurrence sur les mêmes ressources. Les machines virtuelles dynamiques peuvent donc être une approche complémentaire au mécanisme d'équilibre de charge classique. En relâchant la contrainte de ressources statiquement attribuées, il est possible de calculer plus rapidement et plus finement un nouveau partage des ressources entre les processus. Enfin, l'utilisation de machines virtuelles est une solution apte à répondre au problème de la tolérance en panne. On considère que la tolérance aux pannes sera un enjeu important pour les futures machines exaflopiques prévues à l'horizon 2020.

Développement d'environnements de programmation spécifiques

L'exécution d'un programme est intrinsèquement liée au fonctionnement du système d'exploitation et plus particulièrement de son noyau. C'est au noyau qu'il incombe la lourde tâche de répondre aux besoins de l'intégralité des services nécessaires à l'utilisateur. La mise au point d'un noyau robuste et générique permettant de répondre à des besoins hétérogènes est un processus long et complexe. Les risques de failles de sécurité ou d'instabilité dans l'exécution du noyau entrainent une forte réticence à l'ajout de modules noyau destinés au calcul haute performance sur les machines en production. Toutefois, il est possible de modifier le comportement du noyau d'une machine virtuelle sans répercussion sur le fonctionnement du noyau hôte. En effet, les failles de sécurité sont moins problématiques, car la machine virtuelle s'exécute au sein d'un processus UNIX hôte dont les données sont privatisées (cloisonnement de la mémoire virtuelle, remise à zéro des pages...). Il est alors possible de modifier le noyau afin de le rendre

plus spécifique à une application scientifique. Dans un support exécutif parallèle, l'ordonnancement des fils d'exécution et la gestion mémoire sont des mécanismes clés dans l'obtention de performances. Ils pourraient être modifiés pour réduire leur interférence sur le déroulement de l'application. Un exemple de modification de la gestion mémoire a été proposé par Sébastien Valat [VPJ13] : il propose la suppression de la remise à zéro des pages entre deux allocations pour les processus d'un même utilisateur. La mémoire d'une machine virtuelle étant privée à un utilisateur, il n'est pas nécessaire de réaliser cette action pour prévenir une faille de sécurité. D'un point de vue plus général, l'utilisateur d'applications scientifiques n'a pas besoin d'une grande partie des services exécutés par le noyau. Les noyaux légers (ayant moins de services) exécutés au sein de machines virtuelles pourraient améliorer l'exécution d'applications scientifiques.

Constellation de services

À plus long terme, nous pensons que les applications scientifiques auront besoin d'interagir entre elles afin d'être éclatées en une multitude de services spécialisés. Les services seront alors mutualisés entre les applications et aptes à passer à l'échelle. Parmi ces services, on distingue tout particulièrement la gestion des entrées/sorties avec l'analyse in situ des applications. Aujourd'hui, la visualisation est devenue une opération nécessitant autant de ressources que la génération des données qu'elle traite. Elle est donc vouée à être intégrée directement au code de calcul afin d'agréger les données et de relâcher ainsi la pression sur les serveurs d'entrées/sorties. Pour cela, on se tourne vers une analyse in situ qui permet de traiter et de réduire en temps réel les données produites par une application. Ce traitement réalisé en parallèle de l'exécution de l'application n'a pas les mêmes besoins et serait donc exécuté par des processus dédiés aux entrées/sorties. Il permettra d'agréger les informations produites par une ou plusieurs simulations facilitant ainsi le dépouillement des données produites tout en limitant la partie de la machine dédiée aux services (instanciés à la demande). De fait, il faudra alors fournir au sein des supports exécutifs l'interface permettant de mettre en relation le traitement de la visualisation avec l'application s'exécutant sur un cluster. Le déploiement distribué d'instances MPI dans des machines virtuelles que nous avons présentées dans le chapitre 3 peut être une première étape vers la mise en place de requêtes et de communications entre applications MPI. Ces mécanismes permettrons d'enrichir les fonctionnalités MPI comme la création dynamique de processus.

Bibliographie

[ACH+07] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Jr, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[AMD08] AMD-v nested paging. Technical report, Advanced Micro Devices, July 2008.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[BDGR97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco : Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4) :412–447, November 1997.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[CCP+10] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem : Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2 :1–2 :3, New York, NY, USA, 2010. ACM.

[cFC07] Wu chun Feng and K.W. Cameron. The green500 list : Encouraging sustainable supercomputing. *Computer*, 40(12) :50–55, Dec 2007.

[Cre81] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5) :483–490, September 1981.

[Don88] JackJ. Dongarra. The linpack benchmark : An explanation. In E.N. Houstis, T.S. Papatheodorou, and C.D. Polychronopoulos, editors, *Supercomputing*, volume 297 of *Lecture Notes in Computer Science*, pages 456–474. Springer Berlin Heidelberg, 1988.

[DPNJ08] François Diakhaté, Marc Pérache, Raymond Namyst, and Hervé Jourdren. Efficient shared-memory message passing for inter-VM communications. In *Proceedings of the International Euro-Par Workshops 2008, VHPC'08*, volume 5415 of *Lecture Notes in Computer Science*, pages 53–62, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[DYL+12] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11) :1471 – 1480, 2012. Communication Architectures for Scalable Systems.

[EMCS+13] AlexandreE. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel

Lorenz. Ompt : An openmp tools application programming interface for performance analysis. In AlistairP. Rendell, BarbaraM. Chapman, and MatthiasS. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 2013.

[Fly11] M. J. Flynn. Flynn's taxonomy, 2011.

[For94] Message P Forum. Mpi : A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[HN] Tsuyoshi Hamada and Naohito Nakasato. Infiniband trade association, infiniband architecture specification, volume 1, release 1.0,quot ; http ://www.infinibandta.com. In *in International Conference on Field Programmable Logic and Applications, 2005*, pages 366–373.

[HPC13] Hpcg benchmark technical specification. Technical report, Sandia National Laboratories, 2013.

[LML87] Michael J. Litzkow, Matt W. Mutka, and Miron Livny. CONDOR : a hunter of idle workstations. Technical Report TR 0730, University of Wisconsin (Madison, WI US), 1987.

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, August 1998.

[Ope08] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[PJN08] Marc Pérache, Hervé Jourdren, and Raymond Namyst. Mpc : A unified parallel runtime for clusters of numa machines. In Emilio Luque, Tomàs Margalef, and Domingo Benitez, editors, *Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2008.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[RG05] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors : Current technology and future trends. 38(5) :39– ? ?, May 2005.

[Rus78] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1) :63–72, January 1978.

[Tho80] J.E. Thornton. The cdc 6600 project. *Annals of the History of Computing*, 2(4) :338–348, Oct 1980.

[TOP] TOP500 Supercomputer Site.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain K ?gi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5) :48–56, 2005.

[UPC05] UPC Consortium. Upc language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[US12] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 149–160, New York, NY, USA, 2012. ACM.

[VPJ13] Sébastien Valat, Marc Pérache, and William Jalby. Introducing kernel-level page reuse for high performance computing. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, pages 3 :1–3 :9, New York, NY, USA, 2013. ACM.